SAPIENZA
UNIVERSITÀ DI ROMA

FACOLTÀ DI INGEGNERIA

Tesi di Laurea Magistrale in

INGEGNERIA INFORMATICA

Salvataggio e Ripristino Autonomico dello Stato degli Oggetti nei Sistemi di Simulazione Ottimistici

Relatore
Prof. Francesco Quaglia

Candidato
Alessandro Pellegrini

Anno Accademico 2009/2010

Frustra fit per plura quod
fieri potest per pauciora

— *Guglielmo da Occam*

Indice

Elenco delle tabelle

Elenco delle figure

Introduzione

La simulazione parallela ad eventi discreti, denominata *PDES* (Parallel Discrete Event Simulation), è una tecnica ben nota nel campo della ricerca per la progettazione e la valutazione di sistemi complessi costruiti a partire da un'analisi del mondo reale. Alcuni modelli di simulazione richiedono infatti una computazione troppo costosa che necessita di una ingente quantità di risorse di calcolo, reperibile solamente attraverso la parallelizzazione e la distribuzione del carico di lavoro su diversi nodi.

L'esecuzione della simulazione consiste semplicemente nel processamento di *eventi discreti* — ossia di eventi il cui accadimento è istantaneo e la cui durata è impulsiva — ad opera di alcuni *processi logici*, ognuno dei quali modella un oggetto realmente esistente in natura ed evolve nel tempo attraverso continue transizioni di stato, in base ai criteri imposti dallo schema di riferimento.

Per assicurare all'utente del sistema un corretto risultato fornito in output, bisogna definire meccanismi di sincronizzazione tra le varie entità partecipanti alla simulazione. In letteratura un modello di sincronizzazione che è tuttora oggetto di studio è quello *ottimistico*, sul quale viene concentrata l'attenzione nella presente tesi. Tale strategia consente di definire protocolli capaci di sfruttare a pieno le potenziali capacità di esecuzione ottenibili attraverso il parallelismo e permette ai processi logici di eseguire eventi fintanto che ve ne sono a disposizione oppure non vengono percepite *violazioni di causalità* degli eventi che possano compromettere il risultato finale.

Per la gestione di quest'ultimo caso è necessario definire all'interno del sistema una procedura di *rollback*, con cui è possibile ripristinare uno stato precedente della computazione, dal quale si può riprendere la simulazione. Ciò è possibile solamente se sono previste tecniche di salvataggio e ripristino dello stato, tramite le quali è possibile collezionare le configurazioni della simulazione incontrate durante la computazione.

In questo lavoro viene affrontato il problema del ripristino dello stato nei sistemi di simulazione ottimistica, con la proposta di un'architettura di gestione dei checkpoint, progettata secondo il paradigma del *calcolo autonomico*. La proposta è unica, nel senso che affronta contemporaneamente le questioni di trasparenza e performance, offrendo allo stesso tempo le seguenti caratteristiche:

1. le operazioni di salvataggio e ripristino dello stato vengono condotte in maniera del tutto trasparente al programmatore del livello applicativo;
2. lo stato degli oggetti di simulazione può essere distribuito su frammenti di memoria non contigui;
3. il salvataggio ed il ripristino dello stato possono essere eseguiti secondo uno schema incrementale o non incrementale;
4. la selezione dello schema migliore viene effettuata a tempo d'esecuzione, utilizzando un approccio innovativo di modellazione ed ottimizzazione, che si basa sulla capacità di catturare fluttuazioni nelle dinamiche d'esecuzione.

Inoltre, il sottosistema proposto in questo lavoro è stato implementato all'interno di una piattaforma di simulazione ottimistica open source, chiamata *ROOT-Sim*.

Il resto di questa tesi è organizzato come segue. Nel capitolo 1 viene proposta una panoramica sulla storia e sullo stato dell'arte della simulazione parallela e distribuita. Il capitolo 2 offre una descrizione dell'architettura di ROOT-Sim, sottolineando in particolare gli aspetti di interesse per l'implementazione del sottosistema autonomico che presento in questo lavoro. L'architettura autonomica viene mostrata nel capitolo 3. I dati sperimentali per la valutazione della validità della proposta in esame sono riportati nel capitolo 4. Nel capitolo 5 viene riportato un confronto con altri lavori collegati a quello presentato in questa tesi.

Capitolo 1

Cenni sulla Simulazione Parallela e Distribuita

Negli ultimi venticinque anni il *Calcolo Parallelo* si è rivelato un argomento di estremo interesse a livello di ricerca sperimentale, attirando l'interesse degli studiosi sotto svariati punti di vista.

La condivisione delle risorse di calcolo consente infatti la creazione di ambienti virtuali dove l'hardware è dislocato geograficamente in punti remoti, al fine di eseguire una maggiore quantità di calcolo, permettendo di concentrare l'attenzione sull'ottimizzazione della *performance* e dello *speedup*. Si realizza così un sistema trasparente all'utilizzatore che, incurante dei compiti svolti dal sistema, può concentrarsi sulla realizzazione dei *modelli* e sull'analisi dei *risultati*.

Il rapido avanzamento alla base della costruzione di questi sistemi virtuali nasce dalla prorompente esigenza di eseguire simulazioni su vasta scala e negli ambiti più disparati: ingegneria, economia, ricerca militare, biologia e scienza in generale. Simulazioni di questo tipo richiedono, per la loro elaborazione, ingenti risorse di calcolo che possono essere ottenute esclusivamente in ambienti paralleli e distribuiti, dove i vari nodi della rete cooperano al fine di raggiungere un risultato comune, soggetti a requisiti stringenti di velocità d'esecuzione e sincronizzazione, affidabilità della comunicazione, latenza dei messaggi e parallelismo del calcolo.

In seguito alla formulazione del paradigma di calcolo parallelo e distribuito, avvenuta agli inizi degli anni ottanta, la necessità di implementare un'architettura che consentisse il pieno soddisfacimento delle specifiche applicative imposte dall'utente ha spinto i ricercatori verso la definizione di un

sistema multiprocessore distribuito, efficiente e performante, in cui i nodi partecipanti, dotati di risorse di calcolo adatte al tipo di simulazione, coordinano il proprio lavoro con l'intento di ottenere il risultato finale voluto, ricorrendo a meccanismi basati su *memoria condivisa* o comunicazioni di tipo *scambio di messaggi* per mezzo di una rete LAN.

Questo modello di calcolo parallelo prevede che il sistema volto a coordinare lo svolgimento dell'intera simulazione sia in grado di risolvere problemi di sincronizzazione tra i vari nodi attivi, in modo tale che sia possibile creare tra loro accoppiamenti e dipendenze, assicurando allo stesso tempo che le operazioni — processate in parallelo per motivi di efficienza — mantengano lo stesso risultato finale, ottenibile per mezzo di una esecuzione sequenziale[1]. Questo traguardo è raggiungibile esclusivamente se l'esecuzione parallela delle operazioni viene svolta seguendo lo stesso *ordine logico* stabilito dall'esecuzione sequenziale, in cui le operazioni vengono svolte nel pieno rispetto dell'ordine cronologico.

L'indagine nel campo della simulazione parallela e distribuita ha inizio nel 1979 con l'articolo di Chandy e Misra in [CM79]. Il concetto di *PDES* (*Parallel Discrete Event Simulation*), descritto per la prima volta in [Fuj90], nasce come evoluzione del precedente *DES* (*Discrete Event Simulation*). Esso consiste in un paradigma distribuito per l'esecuzione di modelli simulativi.

Con il termine *simulazione* si intende un modello logico-aritmetico capace di rappresentare una caratteristica di un qualunque sistema fisico presente in natura, schematizzabile attraverso algoritmi e/o formule matematiche. Ad ogni simulazione è associato uno *stato globale* che rappresenta la totalità delle informazioni applicative gestite, e un insieme di *eventi* continui o discreti generati progressivamente, che fanno sì che lo stato subisca delle modifiche.

Una simulazione è detta *discreta* quando le operazioni previste dagli eventi avvengono istantaneamente ed hanno una durata impulsiva. In questo progetto mi concentrerò unicamente su questo tipo di simulazioni.

L'idea del PDES, la cui architettura è mostrata in figura 1.1, è quella di concretizzare un programma di simulazione installato su calcolatori paralleli (remoti o locali) e basato sul processamento di *eventi discreti*. Ciascun

[1] Il concetto qui espresso ricorda, in maniera molto forte, quello di *serializzabilità*.

evento può causare modifiche più o meno complesse allo stato globale o parziale della simulazione. Gli eventi sono messi in correlazione temporale da un tempo logico discreto chiamato *timestamp* (o Logical Virtual Time, LVT). Questa correlazione fa sì che sia possibile una forma di sincronizzazione e coordinazione tra tutti i processi partecipanti alla simulazione, così da poter raggiungere un risultato finale comune e corretto.

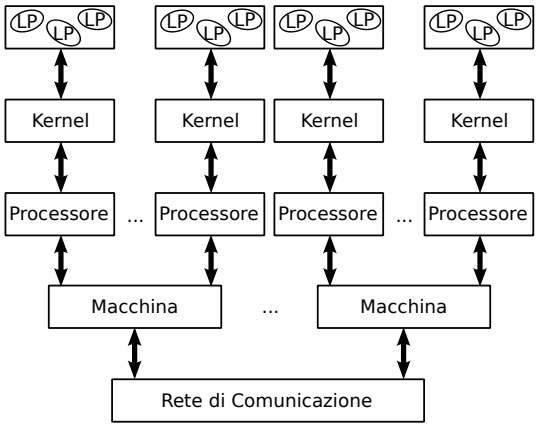

Figura 1.1: Architettura di un sistema PDES

Per quanto riguarda l'applicazione di simulazione discreta, essa consiste in un programma *general purpose*, definito dall'utente in tutti i dettagli di interesse per lui ed in grado di eseguire un qualsiasi tipo di calcolo, purché esso sia riconducibile ad una *catena* di eventi discreti. In particolare il programma simulato comunica con la piattaforma sottostante (chiamato generalmente *kernel di simulazione*) per mezzo di un'interfaccia minimale con cui è in grado di notificare la creazione di un nuovo evento da processare in futuro, eseguire un evento prescelto dal sistema ed esprimere un giudizio su un generico *predicato globale*, che può coincidere, ad esempio, con la verifica della terminazione della simulazione.

Da un punto di vista modellistico la simulazione può essere vista come una collezione di N oggetti detti *logical processes* (LP), indicati con $LP_0, LP_1, \ldots, LP_{N-1}$, a ciascuno dei quali viene associato uno stato, denominato $S_i \subseteq S$, contenente un sottoinsieme di variabili strettamente necessario all'evoluzione della singola istanza della simulazione. L'insieme S corrisponde alla totalità degli stati e quindi raccoglie tutte le informazioni

rilevanti ai fini della simulazione. Durante una generica simulazione ogni LP deve quindi svolgere i seguenti compiti:

- eseguire — in modo parallelo rispetto agli altri LP — gli eventi, in maniera sincrona oppure asincrona;
- generare eventi interni (*local events*), che coinvolgono solamente se stesso, oppure esterni (*remote events*), destinati ad altri LP;
- interagire con gli altri LP attraverso uno scambio di messaggi.

Lo scopo principale di un LP, pertanto, è quello di cooperare con gli altri processi logici in diversi contesti, mediante lo scambio di messaggi di formato predefinito, al fine di raggiungere il risultato di elaborazione desiderato. Ciascun messaggio contiene al suo interno le informazioni necessarie alla gestione di un evento associato ad un particolare timestamp. Il susseguirsi degli eventi comporta un flusso d'esecuzione non deterministico ed impredicibile, non ripetibile in caso di rilancio del programma: questo significa che, pur dovendo la simulazione raggiungere, per considerarsi conclusa correttamente, un preciso stato finale che sia verificabile attraverso la valutazione di un predicato globale, le configurazioni intermedie attraversate possono essere le più disparate.

La tecnica di parallelizzazione del calcolo introdotta con il paradigma PDES, tipica degli ambienti asincroni in cui gli eventi si susseguono ad intervalli irregolari, presenta delle problematiche di difficile trattazione che in parte compaiono anche nel paradigma DES e che negli ultimi anni sono state oggetto di studi ed approfondimenti. Le problematiche fondamentali che un simulatore deve pertanto affrontare sono:

- *coerenza dello stato*: è strettamente necessario mantenere lo stato globale in una forma sempre coerente con la specifica del programma e con lo storico degli eventi eseguiti fino al tempo corrente, così da evitare una transizione verso configurazioni di stato errate o incerte, tali da compromettere il risultato finale dell'esecuzione; ciò è possibile esclusivamente se l'ordine di causalità delle operazioni viene rispettato;
- *scheduling*: la simulazione deve procedere attraverso l'esecuzione di eventi in attesa, generati dall'applicazione e ordinati in base a timestamp crescenti; il compito di stabilire quale LP debba procedere con

l'esecuzione — così da permettere l'evoluzione del proprio stato — e quale sia l'ordine con cui gli eventi in attesa debbano essere processati (nel rigoroso rispetto delle relazioni di causalità) viene affidato ad uno *scheduler*;

- *clock globale*: in un ambiente distribuito ed asincrono, ogni nodo del sistema non è mai perfettamente sincronizzato ed allineato con gli altri, ovvero è completamente assente sia la visibilità di un orologio globale condiviso da tutti, sia la percezione delle sue variazioni da parte di tutti contemporaneamente. Uno strumento del genere è però indispensabile per regolare la cooperazione e rispettare i vincoli di sequenzialità; proprio per questo bisogna escogitare un meccanismo alternativo affinché i nodi che partecipano alla simulazione possano adottare un tempo globale comune, detto *Global Virtual Time*, indispensabile per la coordinazione delle operazioni;

- *scambio di messaggi*: per consentire agli LP di generare eventi remoti, e per far sì che la piattaforma di simulazione possa realizzare delle forme di sincronia tra nodi differenti, è necessario consentire l'interazione tramite un meccanismo orientato allo scambio di messaggi. All'interno di questi messaggi — la cui struttura è definita dal programmatore tramite la specificazione di un formato noto a tutti i componenti dell'architettura — gli eventi vengono imbustati, spediti e ricevuti dal destinatario presso cui devono essere processati. I messaggi ricevuti da un LP vengono memorizzati in una coda FIFO (first-in-first-out) secondo l'ordinamento definito dai loro timestamp, così da preservare la sequenzialità e la correttezza della esecuzione.

  A causa dell'asincronia ed inaffidabilità della rete di comunicazione, potenzialmente soggetta a congestione ed errori di trasmissione, non è garantito che i messaggi inviati da LP_i arrivino a LP_j, né è garantito che l'ordine di inserimento nella rete corrisponda a quello di ricezione; inoltre, a causa della natura distribuita del sistema e della concorrenza nell'esecuzione, è possibile che eventi diversi vengano generati con lo stesso valore di timestamp.

  Questo scenario richiede una gestione accurata ed affidabile delle code dei messaggi, così da evitare di tralasciare un qualche evento durante

l'esecuzione che possa comportare la formazione di uno *straggler*, ossia un messaggio ritardatario[2].

La gestione delle operazioni di scambio di messaggi e di sincronizzazione, così come l'esecuzione di tutte le procedure volte a garantire la correttezza della simulazione, sono affidate ad un *kernel di simulazione*, su cui si appoggia il programma di *livello applicativo*, che contiene le reali specifiche del modello da simulare[3].

Come accennato, il kernel di simulazione si occupa anche di determinare l'ordine con cui ciascun LP dovrà processare gli eventi in attesa (generati dallo stesso LP o ricevuti da un altro). La strategia più usuale è quella di selezionare l'evento E_{min} con timestamp T_{min}. La ragione di queste scelte dipende dal fatto che, qualora venisse eseguito prima un evento E_i con timestamp $T_i > T_{min}$, esso potrebbe modificare lo stato dell'oggetto di simulazione, producendo un effetto che andrebbe ad alterare il processamento dell'evento E_{min}.

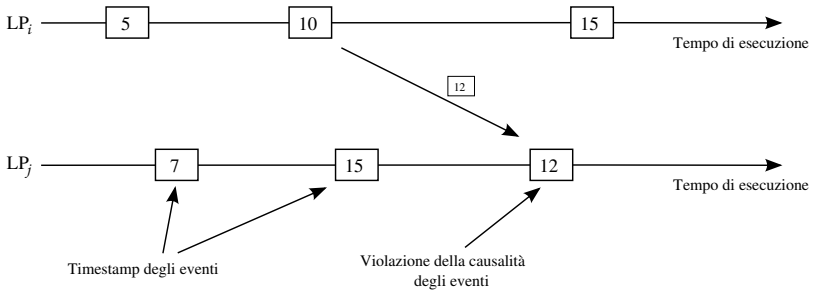

Figura 1.2: Violazione della causalità

Qualora un LP ricevesse, in alcuni contesti, un messaggio contenente un evento con timestamp precedente all'LVT corrente, come si può vedere in figura 1.2, occorrerà adottare tecniche stringenti per ripristinare uno stato precedente ed assicurare l'esecuzione secondo l'ordine corretto.

[2]Propriamente, un *messaggio straggler* è un messaggio contenente un evento il cui timestamp è precedente all'LVT corrente dell'LP coinvolto.

[3]Nel Capitolo 2 verrà presentata più nel dettaglio l'architettura stratificata dei simulatori, qui accennata.

1.1 Strategie di Sincronizzazione

Il paradigma PDES viene implementato secondo diverse strategie ampiamente discusse in letteratura (cfr., [Fuj89], [Fuj93], [Rey88]). Le principali consistono in un approccio *conservativo*, *ottimistico* ed *ibrido*.

Esse sono ancora attualmente oggetto di studio ed in continua evoluzione, comprendono tecniche e strategie basate su differenti punti di vista e presentano una serie di vantaggi e svantaggi che non consentono al programmatore di stabilire in generale una preferenza netta verso un approccio piuttosto che un altro, a meno che non vengano imposti requisiti stringenti sul tipo di simulazione che si intende eseguire.

1.1.1 Strategia di Sincronizzazione Conservativa

Storicamente, l'approccio conservativo è il primo meccanismo di simulazione parallela e distribuita orientata al processamento di eventi discreti. In base a [FN92, Nic96], questa strategia evita gli errori di causalità in fase di simulazione, in quanto si affida ad una strategia mirata a determinare se l'esecuzione di un evento E_i produca un effetto corretto o scorretto sullo stato di un LP.

Nel caso in cui l'evento non comporti alcun rischio, si dice che il processamento dell'evento E_i è *safe*, mentre in caso contrario, quando l'evento successivo nella coda degli eventi in attesa non è quello previsto dall'ordine causale, è detto *unsafe*. Più precisamente, seguendo il carattere conservativo, un processo esegue con successo il successivo evento in attesa E_i se non esiste al momento dello scheduling alcun evento con un timestamp più piccolo di T_i e se il sistema è in grado di stabilire che è impossibile ricevere in un immediato futuro un altro evento E_j con timestamp $T_j < T_i$.

In questo modo viene garantito che l'esecuzione dell'evento non produca una violazione dei vincoli locali e globali di causalità. Questo risultato è ottenibile soltanto se l'ordine cronologico dei messaggi viene preservato su ogni canale di comunicazione tra LP, ovvero attraverso la costruzione di link affidabili gestiti con strategia FIFO. Supponendo che ogni canale trasporti messaggi per un singolo LP, e che ad esso sia associato un clock, ad ogni passo di simulazione viene consegnato all'LP in questione l'evento in attesa con timestamp più piccolo. In questo modo gli eventi in transito su tutti i canali in ingresso vengono processati in base a timestamp non decrescenti.

Nel caso in cui il sistema non possa assicurare queste condizioni con sicurezza, ovvero stabilisca che il prossimo evento in attesa è *unsafe*, esso è costretto ad interrompere il flusso di controllo dell'LP fino a quando non venga ripristinata una situazione di normalità per cui è possibile riprendere il processamento degli eventi (tipicamente questo avviene dopo la ricezione di un qualche messaggio ritardatario). Ovviamente, un approccio di questo tipo può condurre ad un *deadlock*, ossia una situazione indesiderata di stallo in cui tutti gli LP sono in attesa di un qualche evento sbloccante destinato a non sopraggiungere mai, dal momento che tutti sono in attesa.

Proprio per questo motivo sono stati studiati dei meccanismi che consentono la gestione di fenomeni di deadlock, caratterizzati dalla presenza di canali di comunicazione tra due LP con code di eventi vuote e valori di clock relativamente piccoli. La presenza di un ciclo di code vuote presso un sottoinsieme di LP coinvolti in uno scambio di messaggi, ossia la presenza di un ciclo di incrementi nulli di timestamp presso questi LP, denota l'attesa inutile di un qualche evento in input destinato a non avvenire mai, e dunque la formazione di deadlock. Le tecniche principali, elencate in [CM79], sono le seguenti:

- *deadlock avoidance*, con cui si evita completamente una qualsiasi forma di deadlock. Allo scopo di determinare se è possibile processare o meno un messaggio, il sistema deve assicurare che la sequenza dei timestamp dei messaggi inviati consecutivamente da LP_i e memorizzati presso la coda di LP_j sia non decrescente, in modo da garantire che il timestamp dell'ultimo messaggio ricevuto da LP_j e inviato da LP_i costituisca un limite inferiore sui timestamp dei successivi messaggi scambiati tra i due processi logici. Per evitare la formazione di deadlock si utilizza la tecnica dei *null messages* al fine di sincronizzare gli LP e farli uscire dalla condizione di attesa reciproca: un messaggio nullo con timestamp T_{null} viene spedito da LP_i verso LP_j per indicare il fatto che nessun messaggio verrà in seguito spedito da LP_i con timestamp minore di T_{null};

- *deadlock detection*, con cui la transizione verso una situazione di deadlock — anche se erronea — viene permessa dal sistema. Allo stesso tempo, però, sono previsti opportuni meccanismi di sincronizzazione attraverso i quali è possibile uscirne fuori: senza ricorrere ai null message, il deadlock può essere rivelato osservando che l'evento associato

al timestamp più piccolo T_{min} è sempre safe per il processo, pertanto per uscire dal deadlock è sufficiente far processare questo evento;

- *deadlock recovery*, con cui il sistema è capace di ripristinare uno scenario corretto a partire da una condizione di stallo, riconducendolo ad una forma in cui il deadlock non ha più possibilità di ripresentarsi.

La capacità del sistema di determinare se l'evento scelto dal sistema sia safe, ossia non comporti problemi in un immediato futuro, viene detta *lookahead*: se un processo logico LP_i all'istante di tempo T_i può predire con esattezza che tutti gli eventi da lui generati fino al tempo di simulazione $T_i + L$ sono safe, allora il processo in questione ha un lookahead pari a L.

Ogni LP ha sempre $L > 0$, dal momento che il minimo valore di lookahead coincide con il minimo incremento di timestamp tra due eventi generati consecutivamente. A volte però il valore di lookahead di un LP potrebbe risultare troppo basso e rallentare l'esecuzione della simulazione, dato che molti eventi verrebbero giudicati unsafe per evitare il rischio di produrre uno stato incoerente: sono dunque previste alcune tecniche volte ad aumentare il lookahead, basate principalmente sulla precomputazione di una parte della catena degli eventi ancora in attesa, al termine della quale viene effettuata un'analisi dello stato per valutarne la coerenza (cfr. [Nic88]).

Un altro modo per determinare se un evento è safe o meno si basa sul concetto di distanza tra LP, chiamata *object distance* (cfr. [Aya89]); questo termine indica la quantità di tempo di simulazione minima che può trascorrere prima che un evento di un LP non ancora processato possa essere considerato indipendente — direttamente o indirettamente — da un altro LP. In questo modo si può porre un limite sul timestamp degli eventi che in futuro potranno essere ricevuti.

I vantaggi e gli svantaggi introdotti dall'approccio conservativo sono discussi ampiamente in [Nic93, Nic96]. I vantaggi evidenti sono i seguenti:

- *aggressiveless* (assenza di aggressività);
- *riskless* (assenza di rischi);
- *sincronizzazione minima tra processi.*

Per *aggressiveless* si intende un tipo di strategia in cui l'evolversi della computazione non può mai portare ad uno stato di incoerenza, assicurando che ogni passo della simulazione non induca alcuna condizione di errore.

Con il termine *riskless*, invece, si indica un comportamento tale per cui il calcolo di un risultato parziale (o totale) della simulazione da un processo logico LP_i non implica alcun errore.

Inoltre, poiché la computazione procede sempre in avanti (ossia, non si verificano situazioni d'errore che richiedono il ripristino di uno stato precedente), la costruzione di un meccanismo di sincronizzazione tra gli LP è più immediata e comporta un overhead minimo.

Lo svantaggio maggiore dell'approccio conservativo, oltre il fatto di fornire un'esecuzione più lenta e meno performante, è che non è possibile sfruttare al meglio il parallelismo in fase di elaborazione: infatti, se un evento E_i dipende logicamente da un altro evento E_j in maniera diretta (o indiretta), l'approccio conservativo prevede che i due eventi E_i ed E_j vengano eseguiti sequenzialmente come da specifica. Nel caso in cui, invece, non vi sia dipendenza logica, i due eventi potrebbero essere processati contemporaneamente; il sistema forza invece l'esecuzione affinché avvenga in modo sequenziale anche se non è necessario.

Un altro svantaggio risiede nel fatto che, per ottenere una buona performance, occorre avere piena fiducia nel lookahead: gli algoritmi di tipo conservativo tendono infatti a diminuire le capacità di previsione del lookahead, generando facilmente situazioni di blocco che diminuiscono la velocità di esecuzione.

1.1.2 Strategia di Sincronizzazione Ottimistica

Un approccio di *sincronizzazione di tipo ottimistico*, rispetto ad un approccio di *sincronizzazione di tipo conservativo*, sceglie un evento E_i unicamente tra quelli locali, senza preoccuparsi dello stato della simulazione degli altri LP.

L'esempio più significativo di simulazione ottimistica è dato dal meccanismo *Time Warp* presentato in [Jef85]: esso processa comunque gli eventi disponibili, è in grado di rivelare eventuali errori ed interrompere il flusso degli eventi, effettuando il ripristino di uno stato perfettamente coerente dal quale ripartire, tenendo in considerazione le nuove informazioni sull'evento che ha causato questa interruzione. Con questo approccio viene sfruttato

appieno il concetto di parallelismo, poiché non vengono effettuate operazioni di controllo sugli altri LP per verificare se un evento sia safe o unsafe.

In caso di un errore (ossia se si riceve uno *straggler*) è necessario avviare una procedura di *recovery*, attraverso la quale gli effetti di tutti gli eventi eseguiti in modo prematuro vengano cancellati. Completata quest'operazione, si procede con una fase denominata *rollback*: in essa viene eseguito il riprocessamento di tutti gli eventi (compreso quello che ha generato l'errore) rispettando l'ordine temporale. Il numero di eventi da riprocessare prende il nome di *distanza di rollback*.

Poiché l'esecuzione incorretta di questi eventi può aver causato l'invio di messaggi ad altri LP, è prevista una tecnica di invio di *anti-messaggi* che forzano la cancellazione (eventualmente causando rollback) degli eventi negli altri LP. In figura 1.3 viene presentato un possibile scenario di questo tipo.

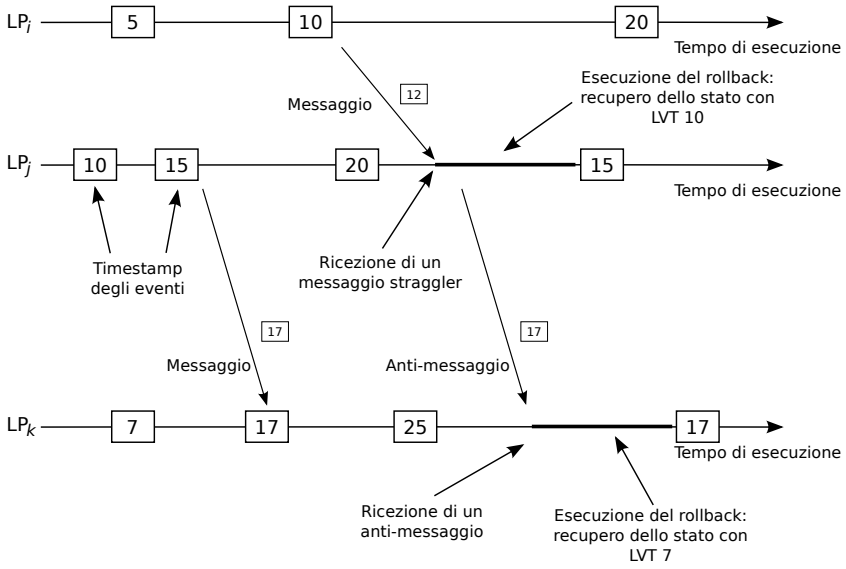

Figura 1.3: Esempio di rollback

In [Jef85] si suggerisce l'uso di un tempo logico comune a tutti gli LP, denominato *Global Virtual Time* (GVT). Esso corrisponde ad un valore di virtual time che costituisce un *lower bound* di ogni futuro rollback e viene calcolato mediante un protocollo distribuito, valutando il minore tra i timestamp degli eventi in attesa su tutti gli LP sparsi sulla rete. Le operazioni

eseguite ad un tempo logico inferiore al GVT possono essere considerate *committed*, ovvero concluse ed impossibili da annullare. Ciò implica che i rollback generati dopo il calcolo del GVT non possono mai essere causati dalla ricezione di straggler con timestamp $T_{straggler} \leq GVT$. In [Bel90] e [LL89] vengono analizzati diversi algoritmi con cui è possibile definire un protocollo distribuito tra i vari nodi della rete che permetta di calcolare periodicamente il GVT, rispettando tutti i vincoli di coerenza.

La tecnica di sincronizzazione ottimistica presenta, tra gli altri, i seguenti vantaggi:

- *elevato grado di parallelismo*: il sistema ottimistico prevede infatti che il processamento di due eventi E_i ed E_j possa avvenire in modo concorrente anche se non si ha la completa certezza che che essi possano davvero essere logicamente indipendenti l'uno dall'altro, contrariamente a quanto viene fatto invece nel caso conservativo, in cui il loro processamento risulterebbe sequenziale;

- utilizzo di *informazioni reali* per ottenere una corretta esecuzione: con questa tecnica non vi è necessità di fare affidamento sul lookahead o sulla object distance per determinare se un evento ancora da processare sia safe o unsafe; è infatti sufficiente che il sistema consulti esclusivamente le informazioni raccolte per stabilire se continuare l'esecuzione oppure avviare un rollback;

- *sincronizzazione trasparente all'applicazione*: il programmatore dell'applicazione non deve preoccuparsi della sincronizzazione tra i vari LP, né della gestione dei checkpoint e dei rollback, né del calcolo del GVT. Egli ha unicamente il compito di definire il codice applicativo, indipendentemente dal lavoro svolto dal sistema sul quale si appoggia;

- *performance* e *velocità*: dal momento che il sistema non interrompe il processo di simulazione di un LP a causa di incertezze circa la sicurezza di un evento, il grado di parallelismo viene aumentato, raggiungendo così livelli prestazionali migliori.

Per quanto riguarda gli svantaggi della simulazione ottimistica, si hanno:

- *aggressiveness*: l'evoluzione della computazione può portare a configurazioni di stato incorrette, a causa del processamento prematuro

di alcuni eventi, obbligando così il sistema a svolgere un rollback che richiede tempo addizionale;

- *risk degree*: un risultato parziale incorretto generato da un LP verso altri processi potrebbe compromettere la loro esecuzione;

- *memory usage*: l'impiego di tecniche di checkpointing per costruire log da cui è possibile ripristinare lo stato precedente ha un impatto notevole sull'utilizzo di memoria da parte del sistema;

- *overhead addizionali*: l'esecuzione di rollback, checkpointing e della sincronizzazione, può portare a degradazioni della performance e dell'efficienza, oltre che — in casi estremi — anche a fenomeni di *trashing*, qualora le risorse di calcolo di un nodo del sistema siano impiegate maggiormente per lo svolgimento di compiti di manutenzione.

Tipicamente, si incorre nel trashing quando si crea un effetto domino dovuto a rollback in cascata, oppure quando si verifica un incremento della distanza di rollback, generando un aumento del tempo speso per completare ciascun rollback[4].

Uno scenario in cui si verifica l'effetto domino è quello in cui i vari LP sono strettamente accoppiati tra di loro, ossia ricorrono ad un fitto scambio di messaggi per generare nuovi eventi; le dipendenze così create obbligano il sistema a svolgere diversi rollback non appena viene rivelato uno straggler, durante i quali vengono generati svariati anti-messaggi che ne scatenano altri.

Un episodio di incremento della distanza di rollback potrebbe invece verificarsi nel caso seguente: si supponga di disporre di due processi logici LP_i ed LP_j, il primo dei quali è in leggero vantaggio sul secondo in termini di unità temporali processate, e si supponga anche che il rollback di un evento richieda di norma più tempo della sua effettiva esecuzione in avanti; assumendo che il secondo induca un rollback sul primo a causa della spedizione di un messaggio straggler, si ottiene che LP_j ha la possibilità di avanzare velocemente con la sua computazione, mentre LP_i è costretto a rallentare lentamente la propria a causa del rollback. Una volta completato il rollback, i due LP procedono parallelamente alla stessa velocità ma, se dovesse presentarsi la situazione in cui LP_i invii uno straggler ad LP_j , si ritorna

[4]Generalmente, si assume che la durata di un rollback sia proporzionale alla sua lunghezza.

esattamente allo scenario precedente, con l'unica differenza che la distanza di rollback potrebbe essere incrementata. Se quindi questo pattern si dovesse presentare frequentemente, il risultato ottenuto determinerebbe una degradazione delle performance a causa della crescita esponenziale del tempo di rollback e della decrescita della velocità di avanzamento nella computazione.

1.1.3 Strategia di Sincronizzazione Ibrida

I due approcci finora illustrati presentano entrambi delle limitazioni quando la complessità e la dimensione della simulazione crescono. Infatti in ambienti *LAPSE* (Large Application Parallel Simulation Environment), in cui vengono eseguite simulazioni su larga scala, la soluzione conservativa appare limitata dal forte overhead dovuto al meccanismo del bloccaggio dell'esecuzione degli eventi unsafe e da un eccessivo affidamento nei confronti del lookahead; la soluzione ottimistica, invece, può vacillare a causa del continuo lavoro di checkpointing e di rollback eseguiti in cascata.

Entrambi gli approcci, pertanto, risultano essere difficilmente scalabili e adattabili a tipologie di utilizzo generico. Per questo motivo, in [RAT93] è stata proposta una combinazione tra l'approccio conservativo e quello ottimistico, così da tentare di compensare, seppur in maniera parziale, le loro limitazioni attraverso un nuovo schema di sincronizzazione. Il risultato di questa ricerca ha reso possibile la creazione di un nuovo sistema di simulazione PDES chiamato *LTW* (Local Time Warp), più adatto all'esecuzione di simulazioni con migliaia di L,P dove è richiesta una maggiore potenza di calcolo.

L'approccio ibrido può essere realizzato seguendo uno di questi procedimenti:

- introduzione della variante ottimistica in un contesto conservativo;
- limitazione della flessibilità riscontrabile con l'ottimismo;
- uso combinato e alternato di entrambi gli approcci.

Per introdurre il paradigma ottimistico in un contesto conservativo è sufficiente fare in modo che il sistema, ogni volta che rivela un evento unsafe di un LP, anziché bloccare la computazione per quel processo logico, lo esegua in modo ottimistico. Tale esecuzione speculativa può così sfociare in due possibili scenari: in un primo caso il risk degree è pari a zero a discapito

di una aggressiveness illimitata, mentre nel secondo caso si può avere una aggressiveness limitata che però comporta anche un risk degree limitato.

Il primo schema è ancora di tipo conservativo perché la computazione speculativa eseguita ha effetti solo localmente; se si verifica un qualche errore è sufficiente fare un rollback locale che vada a ripristinare lo stato corretto senza la necessità di anti-messaggi che, eventualmente, causino rollback in cascata.

Il secondo schema, invece, è orientato al metodo ottimistico, con la differenza che se un qualche errore dovesse manifestarsi, il numero di eventi da annullare sarebbe limitato. Aggiungendo ottimismo al criterio conservativo, si produce dunque un protocollo ibrido che presenta un livello di aggressività decisamente rilassato e meno incidente rispetto al protocollo ottimistico. Esso presenta però degli schemi poco flessibili nei confronti delle configurazioni dinamiche tipiche di simulazioni ampie e complesse.

Così come il metodo conservativo viene reso meno conservativo, allo stesso modo è possibile effettuare dei cambiamenti al paradigma ottimistico per ottenerne uno meno ottimistico. Per risolvere problemi quali rollback in cascata, distanza di rollback e overhead per il checkpointing è infatti sufficiente evitare che lo stato di un particolare LP raggiunga una configurazione che appartenga ad un tempo futuro troppo lontano, ovvero tale per cui il LVT del processo si allontani troppo rispetto al valore del GVT del sistema.

Un risultato del genere è ottenibile imponendo dei limiti sulla *finestra temporale* che determina quali sono quegli eventi che possono essere processati in modo ottimistico, affinché la loro esecuzione non prosegua illimitatamente ma possa arrestarsi ad un certo istante: il sistema, infatti, eseguirà unicamente quegli eventi che ricadono all'interno della finestra temporale.

La limitazione dell'ottimismo implica anche che un possibile rollback generato in futuro non comporti l'annullamento di troppi eventi da rieseguire. La difficoltà maggiore, però, risiede nella scelta opportuna del limite per ottenere un significativo miglioramento della performance: se l'intervallo temporale risulta stretto, i vari LP tendono ad un comportamento di tipo conservativo, in cui sono costretti a sincronizzarsi frequentemente per il calcolo del GVT; al contrario, se l'intervallo è troppo ampio, la risincronizzazione degli LP potrebbe perdere di significato. A tale scopo si fa spesso ricorso a calcoli probabilistici o ad agenti di intelligenza artificiale che permettono la variazione dell'ampiezza della finestra stimando un limite in base al comportamento della simulazione osservato fino al momento del ricalcolo,

ed alla lunghezza della catena di rollback, come ad esempio in [WT09].

Infine l'ultima strategia ibrida consiste in un semplice passaggio, durante l'esecuzione della simulazione, da quella conservativa a quella ottimistica e viceversa. Pur essendo più semplice delle altre, può sembrare attraente ed utile, specialmente quando il comportamento dell'applicazione di simulazione cambia dinamicamente nel tempo, ma richiede un overhead per la raccolta di una serie di informazioni da memorizzare, processare ed esaminare che potrebbe produrre un degrado delle performance. È proprio questo l'approccio che verrà seguito in questo lavoro.

1.2 Salvataggio degli stati

Per effettuare un rollback è necessario ripristinare lo stato degli oggetti di simulazione relativo ad un timestamp precedente. In [CPF99] viene proposta una tecnica di *computazione inversa*, che consiste nell'invertire il flusso dell'esecuzione del software di livello applicativo (in modo automatico o semiautomatico) al fine di ripristinare una configurazione precedente.

Tuttavia, la tecnica di *salvataggio dello stato* (correlata a quella del *ripristino dello stato*) è giudicata più matura ed evoluta. Le versioni più significative di questo approccio sono riportate in [Jef85, Jef90].

Gli aspetti riguardanti il ripristino di uno stato precedente (*modalità* di checkpointing, *periodo* di checkpointing, *risorse impiegate*, tecniche di *cancellazione dei log* non più necessari) non devono coinvolgere in alcun modo il programmatore dell'applicazione: il suo codice non deve contenere alcun riferimento a routine che trattino di salvataggio e ripristino degli stati[5].

Di seguito viene proposta una breve panoramica sulle tecniche di salvataggio degli stati che, nel corso degli ultimi anni, sono state proposte dalla letteratura sul calcolo ottimistico.

1.2.1 Copy State Saving (CSS)

La tecnica più semplice di salvataggio degli stati, presentata per la prima volta in [Jef85], è quella del *CSS* (Copy State Saving). Essa consiste nell'ef-

[5]Tuttavia, nella letteratura sono stati proposti alcuni modelli di programmazione che richiedono allo sviluppatore di fornire informazioni su come effettuare il salvataggio incrementale degli stati.

fettuare una copia completa dello stato[6] di un LP ogni volta che lo scheduler degli eventi determina quale dovrà essere l'evento successivo da eseguire.

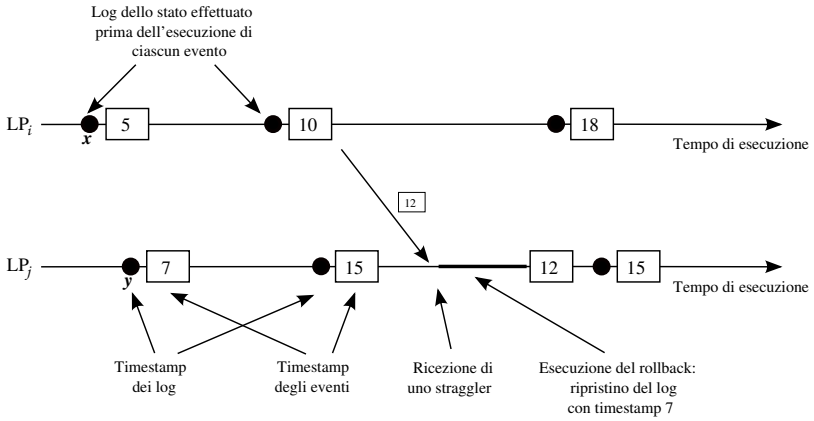

Figura 1.4: Esempio di rollback con CSS

I log vengono marcati con il timestamp relativo all'evento processato immediatamente prima della cattura dello snapshot. In questo modo, qualora sia necessario effettuare un rollback a causa di uno straggler, si potrà determinare con facilità da quale stato occorrerà far ripartire la simulazione (basterà ripristinare il log con l'LVT associato immediatamente minore del timestamp dello straggler).

Poiché, come mostrato in figura 1.4, si avrà un log per ciascun evento, non è necessario riprocessare alcun evento per allinearsi al timestamp dello straggler[7].

Questa tecnica di checkpointing presenta alcuni svantaggi, tra cui l'ingente dispendio di risorse in termini di spazio (per conservare le fotografie degli stati) e di tempo (per eseguire l'operazione di log a ciascun evento).

Dal momento, quindi, che vi è un ingente consumo di memoria, assume un ruolo fondamentale l'operazione di *fossil collection* (cfr. [Jef90]). La fossil collection è quell'operazione che si occupa di eliminare in modo definitivo i log più vecchi, che vengono giudicati inutili per future operazioni di rollback[8].

[6] Tipicamente il sistema, insieme allo stato, effettua una copia anche dei metadati necessari ad un successivo ripristino.

[7] L'operazione di riallineamento, qualora necessaria, prende il nome di *coasting forward*.

[8] L'operazione di *fossil collection* è strettamente legata al concetto di calcolo del GVT espresso nel paragrafo 1.1.2.

1.2.2 Sparse State Saving (SSS)

Per tentare di migliorare la tecnica del CSS, sono state proposte alcune varianti chiamate *SSS* (Sparse State Saving). L'idea alla base di queste modalità di checkpointing è quella di fare delle fotografie degli stati in maniera *sparse*, ossia in maniera non sistematica rispetto al processamento di ciascun evento, ma selettivamente in diversi istanti temporali, con periodo costante (PSS, Periodic State Saving) o variabile (ASS, Adaptive State Saving).

Periodic State Saving

La tecnica del *periodic state saving*, detta anche *state skipping*, viene introdotta per la prima volta in [LL90], ma è stata analizzata più nel dettaglio soltanto in [Bel92]. Questa tecnica tenta di limitare l'overhead del salvataggio dello stato di un LP memorizzandolo con frequenza minore rispetto al numero di eventi processati, mantenendo dei log relativi ad alcuni eventi $E' \subseteq E$.

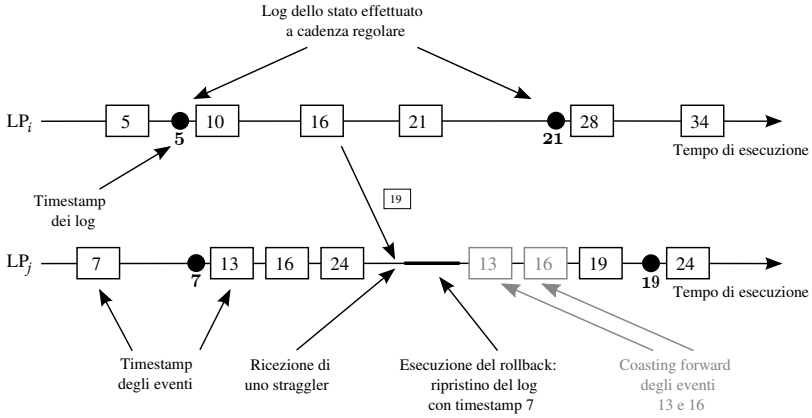

Figura 1.5: Esempio di rollback con SSS

Poiché non si ha più a disposizione un log per ciascun evento, potrebbe verificarsi il caso, mostrato in figura 1.5, della ricezione di uno straggler al tempo T_s non corrispondente ad un istante di checkpoint. Il sistema ricerca allora nella coda degli stati il log più recente e tuttavia inferiore a T_s. Dopo il ripristino di questo stato si effettua il coasting forward (o *state*

rebuilding) che consiste in una veloce riesecuzione di eventi in precedenza già processati (ma di cui si era persa ogni nozione) in *modalità silenziosa*[9].

Nell'esecuzione del coasting forward è necessario garantire che l'esecuzione degli eventi segua la stessa traiettoria seguita in precedenza: in presenza delle stesse condizioni di input e dello stesso stato, il riprocessamento di un evento deve fornire lo stesso output e generare le stesse interazioni con l'ambiente esterno. Questo comportamento viene detto *Piece-Wise-Deterministic* (PWD) ed è stato descritto in [EAWJ96]. Esso è necessario per la corretta ricostruzione dello stato di un LP.

Questa tecnica ha il significativo vantaggio di ridurre il consumo della memoria di lavoro, causando però un aumento di overhead relativo all'operazione di rollback, dal momento che è necessario eseguire il coasting forward. Diventa, quindi, di cruciale importanza determinare un intervallo di checkpointing χ adeguato: se χ è troppo piccolo, si rischia di avere un utilizzo poco efficiente delle risorse di memoria, se è troppo elevato si rischia di indurre un'operazione di coasting forward troppo costosa.

Adaptive State Saving

Lo studio in [PW93] analizza un periodo di checkpointing adattivo per mezzo di un modello basato sul tempo di esecuzione di un LP. Supponendo che durante l'esecuzione di un evento non possa verificarsi *preemption* dovuta a rollback, né alcun invio di messaggi, e supponendo che le lunghezze dei rollback siano indipendenti tra loro, si può individuare l'intervallo di checkpointing ottimale come:

$$\chi_{opt} = \left\lceil \sqrt{\frac{2\delta_s}{\delta_c} + \left(\frac{N}{k_r} + \gamma - 1\right)} \right\rceil \tag{1.1}$$

dove δ_s e δ_c sono i tempi impiegati in media dal sistema per eseguire lo state saving ed il coasting forward, N è il numero totale di eventi committed, k_r il numero di rollback compiuti e γ la lunghezza media di un rollback.

Analogamente e sotto le stesse precondizioni, in [RA94] si propone di osservare durante un periodo T_{obs} il numero di rollback k_{obs} e di eventi eseguiti R_{obs} (sia *committed* che *rolled back*), e di generare una successione

[9] Per *modalità silenziosa* si intende un riprocessamento di eventi che esuli dall'inivio dei messaggi agli altri LP, dal momento che essi sono già stati inviati in precedenza.

numerica di intervalli di checkpoint χ_n, il cui primo elemento (ed i parziali successivi) è dato da:

$$\chi_n = \left\lceil \sqrt{2 \frac{R_{obs}}{k_{obs}} \frac{\delta_s}{\delta_c}} \right\rceil \tag{1.2}$$

I valori saranno calcolati in accordo con il seguente pseudocodice:

if $n = 0$ **then**	$\chi_n \leftarrow \chi_{init}$
else if $k_{obs} = 0$ **then**	$\chi_n \leftarrow \lceil (1-\rho)\chi_{n-1} + \rho\chi_{max} \rceil$
else	$\chi_n \leftarrow max(1, \lceil (1-\rho)\chi_{n-1} + \rho min(\chi_{min}, \chi_{max}) \rceil)$

dove il valore $\rho \in (0, 1)$ determina se si sta dando più peso allo storico di χ_n piuttosto che alle misurazioni correnti; χ_{min} e χ_{max} definiscono dei limiti alle variazioni del periodo.

In alternativa, in [SR96] viene fornita un'ulteriore variante del precedente schema, chiamata *ESSS* (Event Sensitive State Saving), che tiene in considerazione il comportamento variabile degli eventi per selezionare opportunamente il valore χ_{opt}. Questa tecnica mette in risalto il fatto che conviene effettuare il salvataggio dello stato di un LP solamente quando la granularità degli eventi tende ad aumentare, ovvero quando si è in presenza di eventi con lunghi tempi di coasting forward. Pertanto, partendo dal modello presentato in [RA94] e classificando gli eventi in N classi, la piattaforma di simulazione raggruppa nella stessa classe $n \in N$ tutti gli eventi che esibiscono un comportamento simile, discriminandoli in base al loro tempo di coasting forward e selezionando così un valore χ in funzione della classe più ricorrente. Supponendo dunque che ad ogni famiglia di eventi sia associata una *event frequency* f_i e una probabilità di salvataggio dello stato p_i, il valore stimato del periodo di checkpointing ottimale viene calcolato come media geometrica tra tutte le categorie:

$$\chi_{opt} = \left(\sum_{i=1}^{N} p_i f_i \right)^{-1} \tag{1.3}$$

Allo scadere del periodo di checkpointing così identificato, il sistema esegue unicamente il salvataggio dello stato di quegli LP che hanno processato eventi con alta granularità, ovvero appartenenti alla classe con valori elevati di probabilità p_i. In questo modo si riduce l'overhead dovuto al coasting for-

ward, dal momento che si previene la formazione di lunghe catene di eventi da rieseguire.

Nell'esposizione in [FW95] viene invece individuato un approccio totalmente differente rispetto a quelli finora presentati, in cui la regolazione dell'intervallo di checkpointing viene implementato attraverso un meccanismo molto più semplice. La tecnica in questione fa ricorso ad un *algoritmo euristico* che calcola periodicamente la seguente funzione di costo:

$$E_c = C_{ss} + C_{cf} \tag{1.4}$$

in cui C_{ss} e C_{cf} corrispondono rispettivamente ai tempi di CPU impiegati dal sistema per eseguire il salvataggio dello stato ed il coasting forward. Prendendo in considerazione dei periodi di osservazione completamente indipendenti da quelli di checkpointing, il sistema calcola il valore di E_c, proponendosi di riadattare di volta in volta in maniera intelligente il valore di χ_{opt}, aumentandolo di una unità (fino al raggiungimento di una soglia massima) qualora il sistema non riveli significativi incrementi di E_c, o viceversa decrementandolo. Se dunque il comportamento dell'applicazione subisce delle variazioni che causano l'aumento o la diminuzione della probabilità di rollback, ciò viene riscontrato in una variazione della funzione di costo, che si traduce in un'alterazione del periodo di checkpointing.

Un'altra variante presentata in [Qua98] propone di osservare lo *storico degli eventi* di un LP, tenendo in considerazione l'incremento di timestamp tra due eventi consecutivi, per stabilire qual è il momento più conveniente per procedere con l'acquisizione del checkpoint: il rollback può capitare in un momento qualsiasi della computazione, in particolare si verifica in un intervallo I compreso tra i valori di timestamp di due eventi consecutivi. Supponendo dunque che lo stato corrente di un LP abbia raggiunto il tempo logico LVT^* e che il prossimo evento da eseguire stabilito dallo scheduler abbia un timestamp T^*, se l'intervallo di durata $T^* - LVT^*$ che intercorre tra i due eventi consecutivi ha una varianza positiva significativa rispetto al valore medio, vi sono rischi maggiori che si possa incorrere in un rollback, dunque è opportuno memorizzare un log. Senza dunque ricorrere ad onerosi calcoli statistici, è possibile individuare con facilità il momento in cui conviene eseguire il salvataggio dello stato, non appena viene generato

dall'applicazione l'evento successivo.

Le caratteristiche che accomunano le varie teorie descritte portano tutte alla memorizzazione di un log solo quando esso è realmente necessario: il sistema ha la possibilità di eseguire le routine di checkpointing meno frequentemente, riducendo così drasticamente l'overhead dovuto al log. Inoltre, dato che nella successione delle attività del sistema il checkpoint precede, con grande probabilità, un rollback, si hanno maggiori possibilità che il log abbia un valore di timestamp $T_{log} \leq T_{straggler}$ molto vicino a $T_{straggler}$, permettendo così una ricostruzione breve e veloce dello stato mancante.

Un sistema che adotta ASS può riscontrare però anche una serie di penalità che inducono nuovi overhead, dovuti alla collezione di statistiche e ad un calcolo potenzialmente inesatto del periodo di checkpointing, che deve essere comunque gestito per evitare lunghe catene di coasting forward. Il lavoro presentato in questa tesi ha, tra i vari obiettivi, quello di tenere in considerazione errori di misurazione dei parametri e fluttuazioni nei valori osservati, proprio al fine di tentare di ridurre il più possibile questi overhead secondari.

1.2.3 Incremental State Saving (ISS)

Per ovviare al problema della gestione della memoria, in uno scenario in cui siano presenti un alto numero di LP con degli stati di grande dimensione, ed evitare quindi il trashing delle risorse (con un conseguente calo di performance dell'intero sistema) si può adottare un approccio incrementale (ISS), per cui il sistema, invece di salvare l'intero stato di ogni oggetto, salva soltanto le aree modificate dall'ultimo restore, ossia quelle a cui si accede in scrittura.

Questa strategia ha lo scopo di limitare l'incidenza dell'overhead causato dal checkpointing, riducendo il tempo della sua esecuzione e la quantità di memoria consumata dal salvataggio dei dati. Le possibili varianti implementative del paradigma ISS sono:

- *Optimized Incremental State Saving*, la prima versione proposta di ISS;
- *Incremental State Saving con protezione della memoria*, che fa utilizzo di meccanismi di protezione delle pagine allocate per lo stato, sfruttando meccanismi a livello del kernel del sistema operativo;

- *Transparent Incremental State Saving*, che rende possibile l'incrementalità del log per mezzo dell'overloading e della programmazione object oriented;
- *Automatic Incremental State Saving*, che effettua il salvataggio incrementale sfruttando l'instrumentazione del codice applicativo.

Optimized Incremental State Saving

La prima proposta di checkpointing incrementale è stata presentata in [BS93], ed ha aperto il dibattito facendo nascere delle varianti volte ad ottimizzare l'uso della memoria ed a minimizzare l'overhead dovuto al salvataggio dello stato. Questa tecnica prevede che ad ogni evento E_i processato dal sistema venga associata una serie di informazioni costituita da:

- il valore della variabile di stato modificata dopo che l'evento è stato eseguito;
- il valore della variabile di stato prima che l'evento imponga il suo effetto sullo stato;
- il tempo logico T_{gen} in cui l'evento viene generato e quello T_{exe} in cui deve essere eseguito, con $T_{exe} > T_{gen}$;
- l'azione svolta durante il processamento dell'evento.

Le strutture dati gestite dal sistema in questo caso non sono più semplici code di stati salvati, bensì una lista di modifiche (per ciascun LP) strettamente accoppiate agli eventi che le hanno generate. Queste informazioni vengono inserite in una struttura dati che viene gestita per mezzo di una coda FIFO, in cui il sistema memorizza tutti gli eventi eseguiti (e quelli ancora da eseguire) per avere traccia dell'evoluzione dello stato.

La *coda degli eventi* gioca quindi un ruolo fondamentale in quelle situazioni in cui è necessario ripristinare uno stato coerente, dal momento che, per ridurre l'overhead, viene impiegata anche come coda per il salvataggio dello stato, proprio a causa della presenza di meta-informazioni che aiutano il sistema nella fase di rollback. Infatti, ogni volta che un nuovo evento viene processato, la variabile da sovrascrivere è copiata in questa struttura prima della sua reale modifica. Come è noto, uno straggler corrisponde ad un messaggio che viene ricevuto da un LP con timestamp $T_{straggler} < LVT$:

poiché il log è stato preso durante la computazione in maniera incrementale, il rollback procede in modo diverso da quello finora presentato:

- il ripristino di uno stato coerente viene realizzato scandendo in avanti la coda degli eventi a partire dall'evento con timestap $T_{straggler}$, che è il tempo al quale occorre riallineare lo stato stesso, così da poter riprendere la simulazione. Ciò viene fatto ripristinando in ordine cronologico il valore di ogni variabile memorizzata in corrispondenza del processamento dell'evento E_i, con timestamp T_i, tale che $T_{straggler} \leq T_i \leq LVT$, facendo però attenzione ad aggiornare la medesima variabile una sola volta. In caso contrario, la coerenza dello stato non sarebbe garantita.

- per quanto riguarda invece il ripristino della coda degli eventi, è sufficiente rimuovere dalla coda tutti quegli eventi che al tempo logico $T_{straggler}$ non erano ancora stati generati, ovvero quelli con timestamp $T_{gen} > T_{straggler}$;

- una volta completato il rollback, è sufficiente riprendere la simulazione a partire dal tempo logico associato allo straggler, imponendo $LVT = T_{straggler}$.

Questa tecnica, però, ha il forte svantaggio di non essere trasparente al programmatore, in quanto la copia di ogni singola variabile di stato prima della sovrascrittura viene commissionata all'utente, che è dunque costretto ad accedere alle strutture dati della piattaforma sottostante.

Incremental State Saving con protezione della memoria

Un'altra possibile implementazione di ISS ricorre alla protezione degli accessi alle pagine dello spazio di indirizzamento di un processo offerta dai sistemi operativi UNIX. Un suggerimento riguardo l'utilizzo di tale tecnica viene presentato in [SQ05a], anche se il contesto in cui essa viene applicata è ben diverso.

Poiché l'applicazione non deve assolutamente preoccuparsi del salvataggio dello stato e del rollback, ed allo stesso tempo il sistema sottostante non può conoscere la forma dello stato di un LP[10], nel momento in cui un

[10] Il modello di programmazione è infatti *general purpose*, pertanto lo stato può assumere una forma qualsiasi.

singolo campo dello stato viene modificato, il sistema memorizza la pagina acceduta in scrittura dall'applicazione che contiene il campo stesso. Poiché però la piattaforma non ha alcun controllo sugli accessi in lettura/scrittura degli oggetti, per poter raggiungere questo risultato è necessario sfruttare dei servizi offerti dal sistema operativo, come ad esempio la system call `mprotect()` in UNIX, che offre la possibilità di proteggere pagine di memoria. Proteggendo, dunque, tutte quelle pagine che contengono lo stato degli LP, nel momento in cui l'applicazione tenterà un aggiornamento dello stato, il sistema operativo genererà un segnale che potrà essere intercettato dalla piattaforma di simulazione, che verrà dunque avvisata di un tentativo di scrittura. Supponendo che il sistema conosca l'esatta collocazione dello stato di ogni LP all'interno della memoria di lavoro, i passi principali che vengono svolti per effettuare il checkpoint sono quelli tipici della tecnica del *copy-on-write*:

- la piattaforma di simulazione protegge lo stato del singolo LP tramite la system call `mprotect()`;
- l'accesso in scrittura da parte dell'applicazione che vuole modificare lo stato di un LP viene impedito dal sistema operativo grazie al meccanismo di protezione e segnalata alla piattaforma di simulazione, mentre eventuali accessi in lettura vengono permessi perché non costituiscono perdita di informazione;
- il segnale generato viene catturato dalla piattaforma e gestito tramite un *handler*, il cui compito è quello di eliminare la protezione, salvare la pagina contenente la porzione di stato che vuole essere aggiornata (marcandola con un valore di timestamp), ed eseguire la scrittura vera e propria, aggiornando così lo stato;
- successivi accessi in scrittura possono essere trattati in maniere differenti: la protezione delle pagine può infatti essere ripristinata immediatamente dopo l'aggiornamento, oppure allo scadere di un periodo prefissato.

Questa soluzione, oltre a consentire di salvare porzioni ridotte dello stato degli oggetti di simulazione, permette anche di sfruttare il principio di località spaziale della memoria. Allo stesso tempo, però, ha lo svantaggio di essere una tecnica conveniente solo nel caso di stati di dimensioni che

superano decisamente la soglia di 4 KB[11]. Inoltre, essa aggiunge un overhead non indifferente — a causa del coinvolgimento del sistema operativo[12] e della grande quantità di tempo richiesta dalla protezione delle pagine — che dipende anche dalla dimensione dello stato, dalla gestione dei segnali e dalla copiatura delle pagine.

Per coinvolgere il meno possibile il sistema operativo, è opportuno che il sistema adotti una variante *semi-incrementale* dell'approccio, proteggendo unicamente quelle pagine che vengono accedute in scrittura più frequentemente — assumendo che esse siano una minoranza, rispetto al numero totale di pagine che compongono lo stato. Per tutte le altre pagine, il sistema può adottare un'altra tecnica di salvataggio dello stato. Utilizzando questo criterio, comunque, resta la difficoltà di determinare quali siano le pagine accedute in scrittura più frequentemente.

Transparent Incremental State Saving

Questa tecnica, presentata in [RLAM96], propone un'implementazione del checkpointing incrementale tramite l'utilizzo di meccanismi di *overloading* propri della programmazione *Object Oriented*. Ciò restringe la possibilità di applicazione di questa tecnica unicamente a quelle piattaforme (ed a quei programmi applicativi) realizzati con tecnologie Object Oriented, come ad esempio in C++ o Java.

L'idea al centro di questo approccio parte da due punti principali:

- ridefinizione, tramite *overloading*, di operatori, funzioni e metodi utilizzati dal programmatore per processare le varie tipologie di evento;
- *incapsulamento* dei dati corrispondenti alle variabili di stato all'interno di particolari `struct` gestite dal sistema in modo diverso, rispetto alle altre.

Tramite l'incapsulamento, la piattaforma può discriminare quali siano le variabili di stato di un LP, e quali invece le variabili locali utilizzate temporaneamente per il processamento di un evento.

L'applicazione dunque dichiara al sistema le proprie variabili di stato ricorrendo ad una particolare segnatura diversa da quella standard, ad esempio `State<class T>`, con cui si effettua il wrapping di una variabile di classe

[11] 4 KB è la dimensione di una pagina nei sistemi convenzionali.

[12] I continui passaggi da *kernel-mode* a *user-mode* e viceversa (che hanno un costo elevato) diventano troppo frequenti durante la simulazione.

`T` (tipo di dato primitivo) che il sistema deve riconoscere come parte dello stato. Tutti i dati definiti con questa segnatura costituiscono, nel loro insieme, l'intero stato di un processo, i cui campi possono essere manipolati esclusivamente attraverso nuovi operatori, funzioni e metodi ridefiniti tramite overloading di quelli standard. Nella ridefinizione, vengono aggiunte delle operazioni che, prima dell'aggiornamento di una variabile, ne consentono il salvataggio tra le strutture dati del sistema. La classe `State<class T>` è dunque un supporto esterno, nel senso che l'utente non si preoccupa né della sua definizione né dell'overloading dei suoi metodi, pur dovendo egli essere comunque consapevole del fatto che ogni variabile di stato deve essere riferita per mezzo di una istanza `State`, affinché il sistema possa gestirne le modifiche.

È questo, dunque, uno degli aspetti meno convenienti di questa tecnica: pur non dovendosi preoccupare del salvataggio dello stato né del rollback, l'utente deve fornire manualmente alcune informazioni alla piattaforma, che non è pertanto completamente trasparente. Inoltre, in caso di oggetti complessi, non è ancora chiaro come possa la piattaforma inserire codice ad hoc all'interno dei metodi ridefiniti tramite overlaoding al fine di effettuarne il log, dal momento che non è a conoscenza della loro struttura. La trasparenza potrebbe completamente scomparire se fosse il programmatore dell'applicazione ad occuparsene.

Automatic Incremental State Saving

Un'altra implementazione di ISS è presentata in [WP96], ed è di particolare interesse poiché presenta il maggior numero di similitudini, tra gli articoli proposti nella letteratura, con l'architettura e la tecnica presentata in questa tesi.

Essa si basa sull'instrumentazione del file oggetto ottenuto dalla compilazione del codice applicativo, al fine di individuare soltanto quelle parti dello stato che sono coinvolte nel log. Al contrario della tecnica precedente in cui era ottenibile solo un livello di trasparenza parziale, questo metodo è più orientato al programmatore, dato che egli può scrivere la propria applicazione senza doversi preoccupare di inserire nel codice operazioni che si occupino del salvataggio manuale delle variabili di stato prima della loro sovrascrittura.

Il fine ultimo di questa tecnica è quello di lasciare inalterato il metodo e gli strumenti di programmazione standard dell'utente della piattaforma. I

progettisti del sistema mettono a disposizione dell'utente un tool in grado di modificare il file oggetto del codice applicativo, in modo tale che il salvataggio dello stato venga eseguito automaticamente in corrispondenza di una qualsiasi operazione di scrittura. Agendo a livello delle istruzioni macchina, ogni volta che il tool individua all'interno del codice un'operazione di scrittura in memoria, aggiunge prima di essa una `call` ad una subroutine in assembly che si occupa del salvataggio del contenuto della locazione interessata in una nuova area di memoria riservata.

1.3 Ottimizzazioni nel protocollo di checkpointing

In letteratura esistono ulteriori tecniche, che nascono spesso da modifiche apportate a quelle principali finora esposte, che è bene comunque presentare, così da fornire al lettore un quadro più completo sulle modalità utilizzabili per effettuare il salvataggio degli stati.

1.3.1 Tecniche ibride di state saving

La tecnica dello Sparse State Saving, presentata nel paragrafo 1.2.2, negli anni è stata rimodellata in modo da ottimizzare il checkpointing e ridurre il più possibile l'overhead causato dai frequenti salvataggi degli stati che vengono eseguiti durante la simulazione.

In particolare, è stata avanzata in [GUCF97] la proposta di utilizzare una strategia ibrida, chiamata *MSS* (Multiplexed State Saving) che definisca un protocollo di checkpointing che sfrutti i meccanismi PSS ed ISS, presentati nei paragrafi 1.2.2 e 1.2.3, tentando di ridurre i ritardi dovuti ai rollback, gli effetti domino ed il trashing delle risorse.

Le ipotesi sottostanti il MSS sono:

- il tempo di CPU utilizzato per ripristinare uno stato corretto attraverso una lista di log è verosimilmente inferiore rispetto a quello necessario per ricostruirlo a partire da una configurazione salvata;
- l'operazione di coasting forward potrebbe avere un costo eccessivo rispetto a quella di rollback se l'intervallo di checkpoint è troppo ampio e la distanza di rollback è ristretta a pochi eventi da rieseguire;
- a partire dallo storico degli eventi di un LP, il sistema può ricostruire un qualsiasi stato avendo a disposizione una sua fotografia antecedente.

Ogni LP mantiene tra le proprie strutture dati una lista concatenata, nella quale vengono memorizzati i checkpoint eseguiti periodicamente che fanno riferimento esclusivamente alle variabili di stato aggiornate. Inoltre la lista dei log è strettamente accoppiata a quella degli eventi processati dall'LP: ciascun evento è associato al checkpoint con timestamp $T_{log} \leq T_{ev}$, che è in possesso delle informazioni necessarie ad un corretto ripristino dello stato, nel caso in cui l'evento associato dovesse essere annullato. Qualora si verificasse un rollback, quindi, il ripristino dello stato al tempo logico T avviene ripristinando dapprima la porzione di stato ottenuta dal checkpoint con timestamp $T_{log} \leq T$; in seguito si procede a ritroso lungo la lista concatenata, recuperando gli altri frammenti di stato modificati, ricavati dagli altri checkpoint parziali.

La metodologia MSS porta benefici principalmente nelle applicazioni di simulazione ottimistiche interattive, con frequenti operazioni di input e di output richieste all'utente. Inoltre il suo impiego è maggiormente indicato nel caso di simulazioni con lunghi periodi di rollback, dato che ha dimostrato di avere la capacità di migliorarne la velocità di esecuzione.

In [Qua99] viene fornita un'ulteriore ottimizzazione della tecnica dell'Adaptative Sparse State Saving, che si avvale di una combinazione delle tecniche di Periodic State Saving e Probabilistic State Saving. Questa ottimizzazione prevede che il valore del periodo di checkpointing χ venga modificato tenendo in considerazione i parametri misurati a runtime.

Ricorrendo ad una tecnica simile a quella presentata in [FL95] è possibile determinare se in un successivo istante della computazione si possa incorrere o meno in rollback. Il sistema può pertanto variare a proprio piacimento il periodo di checkpointing, tentando di prendere un log esattamente prima che si possa verificare un rollback.

Qualora la probabilità di rollback misurata sia estremamente bassa, invece, il sistema non si limita a non prendere alcun log: un rollback accidentale causerebbe infatti un altissimo costo di coasting forward. In casi come questo, il sistema mostra un comportamento di tipo PSS, memorizzando periodicamente lo stato scegliendo un periodo χ_{max}.

Partendo da questo risultato, in [Qua01] è stato proposto un modello di costo per la selezione ottimizzata della posizione del checkpoint di uno stato. Questo modello permette di determinare se sia conveniente o meno

avviare l'operazione di log prima dell'esecuzione di un qualche evento E, ancora prima che esso modifichi lo stato S dell'applicazione. L'euristica che guida questa procedura tende a minimizzare la lunghezza delle catene di rollback: il sistema decide di pagare, in un determinato istante della computazione, il costo di un checkpoint soltanto se la stima del costo di un suo possibile ripristino futuro ha un valore superiore. Per prendere questa decisione, considerati i seguenti parametri:

- la posizione dell'ultimo checkpoint dello stato S nel tempo logico;
- la granularità degli eventi eseguiti nell'intervallo di tempo logico tra l'ultimo checkpoint compiuto e l'istante corrente;
- la probabilità che lo stato S debba essere in futuro ripristinato a causa dell'occorrenza di un qualche rollback.

che vengono combinati nell'equazione:

$$CR(S) = \begin{cases} \mu_s + P(S)\mu_s & \textbf{se } S \text{ viene salvato} \\ P(S)\left[\mu_s + \sum_{e \in E(S)} \mu_e\right] & \textbf{altrimenti} \end{cases} \tag{1.5}$$

dove:

μ_s è il costo del salvataggio e del ripristino dello stato S, dipendente dalla taglia e dalla velocità della macchina adottata per eseguire il calcolo;

μ_e è la granularità dell'evento e, ossia il tempo di esecuzione necessario per rieseguire un evento che deve essere riprocessato durante il coasting forward;

$P(S)$ è la stima della probabilità che lo stato in questione venga ripristinato in seguito all'occorrenza di un rollback, che dipende dal comportamento dinamico dell'applicazione e dalla lunghezza dell'intervallo $I(S)$, che va dall'istante di tempo in cui è stato eseguito l'ultimo checkpoint fino al timestamp dell'evento e corrente che causa la transizione di stato;

$E(S)$ è l'insieme degli eventi processati nell'intervallo $I(S)$ che devono essere rieseguiti per riallineare lo stato S.

L'equazione 1.5 viene calcolata nelle due forme — come se si dovesse eseguire il salvataggio dello stato e come se ciò non fosse necessario — prima di

eseguire un qualsiasi evento E. Valutando quale dei due risultati è il più conveniente, il sistema può decidere se effettuare o meno il checkpointing dello stato S dell'LP corrente.

1.3.2 Tecniche asincrone di state saving

Il concetto di checkpointing asincrono, presentato in [QS01], si basa sulla possibilità di avere risorse di calcolo sfruttabili parallelamente al flusso di controllo della simulazione, operando in multithreading.

Secondo quanto esposto in [D'A07], il problema può essere affrontato utilizzando lo *shadowing*. In pratica, alle risorse di calcolo dei nodi del sistema distribuito che sono libere dal carico applicativo può essere assegnato il compito del salvataggio dello stato. È necessario, dunque, sdoppiare ogni singolo processo logico creandone una copia *shadow* chiamata SLP (Shadow Logical Process), il cui compito è quello di effettuare i checkpoint durante la simulazione.

La *shadow copy* del processo logico, durante la simulazione, esegue gli eventi nello stesso ordine[13] del processo logico principale[14], occupandosi però in aggiunta di eseguire il salvataggio dello stato. Le operazioni dello *shadow thread*, pertanto, sono asincrone rispetto a quelle compiute dal thread principale: pur mantenendo lo stesso ordine di esecuzione degli eventi, non è dato sapere a priori il ritardo con cui esso eseguirà gli eventi stessi.

Qualora si incorra in un rollback, è necessario garantire una forma di sincronizzazione tra i due thread: quello principale, infatti, ha la necessità di chiedere a quello secondario in quale punto della memoria il log è stato salvato. Tuttavia, poiché la simulazione può evolvere più velocemente, grazie al fatto che le operazioni di salvataggio dello stato sono demandate al thread secondario, si nota un incremento di prestazioni, dovuto anche ad un utilizzo migliore della località dei dati e, quindi, ad una diminuzione dei *cache miss*.

1.4 Riflessioni sulle tecniche esistenti

Le tecniche di sincronizzazione ottimistica studiate per il protocollo Time Warp hanno dato lo spunto per lunghe discussioni nell'ambito di conferenze

[13] L'importanza dell'esecuzione degli eventi nello stesso ordine risiede nel fatto che, alcune applicazioni, possono esibire un comportamento non Piece-Wise-Deterministic.

[14] L'ordine degli eventi è, comunque, deciso dal Kernel di simulazione.

internazionali. Le tematiche affrontate spaziano lungo varie direzioni, con il fine ultimo di produrre una soluzione efficiente per la problematica più critica della simulazione ottimistica: il salvataggio e ripristino dello stato di simulazione.

Non esiste ancora una soluzione ottima: la grande varietà di metodologie consente al progettista della piattaforma di scegliere l'approccio più indicato a supportare l'applicazione che è necessario eseguire. Il punto debole delle tecniche finora proposte risiede nel fatto che esse si basano su ipotesi generalmente vincolanti e restringenti: lo studio dei casi di rollback, infatti, è ampio e complesso, poiché il modello di simulazione proposto è estremamente generale e poiché è strettamente dipendente dal comportamento dell'applicazione e dai ritardi nella rete di comunicazione. Questa variabilità fa sì, pertanto, che una particolare soluzione sia ottima in un determinato scenario, ma meno efficace in un altro.

Una caratteristica importante di tutte le tecniche proposte è quella di tentare di mascherare il più possibile l'approccio ottimistico utilizzato dal sistema, adoperandosi per soddisfare la correttezza della computazione, la rapidità nella produzione dell'output finale, garantendo all'utente la trasparenza maggiore possibile ed una grande flessibilità nella scrittura del programma applicativo. Nella tabella 1.1 sono raggruppate, in breve, tutte le caratteristiche offerte dalle tecniche di salvataggio dello stato finora presentate.

Come anticipato, lo scopo di questo lavoro è quello di presentare due tecniche alternative di checkpointing, mostrando come, con tecniche avanzate di programmazione, è possibile far sì che la piattaforma modifichi da sé il suo comportamento, partendo dalla misurazione di alcuni valori, nel tentativo di ottimizzare al massimo le prestazioni. Parallelamente, si vuole aumentare il livello di flessibilità e di trasparenza di programmazione offerto all'utente finale, nel tentativo di disaccoppiare in maniera netta il sistema dal livello applicativo. Le tecniche discusse finora, infatti, pur offrendo tutti gli elementi necessari a consentire il ripristino dello stato di un LP in una forma coerente, non tengono in alcuna considerazione la possibilità che tale stato possa evolvere nel tempo logico, non solo modificando il proprio contenuto, ma anche crescendo o diminuendo di dimensione.

Il comportamento di un oggetto che preveda l'allocazione e la deallocazione dinamica della memoria è in realtà del tutto ragionevole, in quanto una piattaforma PDES dovrebbe essere in grado di eseguire applicazioni

State Saving	Fossil Collection	Coasting Forward	Checkpointing Periodico	Periodo Adattativo	Stato Dinamico	Incrementalità	Granularità Arbitraria	Trasparenza	Instrumentazione	Granularità degli Eventi	Multiplexing	Modello Analitico	Misure Temporali	Euristiche	Autonomicità	Stabilità alle Fluttuazioni
[Jef85]																
[Jef90]	✓															
[Bel92]	✓	✓	✓													
[PW93]	✓	✓	✓	✓								✓	✓			
[RA94]	✓	✓	✓	✓								✓				
[SR96]	✓	✓	✓	✓						✓		✓				
[FW95]	✓	✓	✓	✓									✓	✓		
[Qua98]	✓	✓	✓	✓						✓		✓				
[BS93]	✓	✓				✓	✓									
[SQ05a]	✓	✓				✓		✓								
[RLAM96]	✓	✓				✓										
[WP96]	✓	✓	✓	✓		✓		✓	✓				✓			
[GUCF97]	✓	✓		✓		✓					✓					
[Qua01]	✓	✓	✓	✓						✓		✓				

Tabella 1.1: Tabella comparativa delle modalità di state saving

general purpose.

Inoltre, nessuna delle tecniche presentate finora prevede un alto grado di variabilità delle dinamiche di esecuzione delle applicazioni. Anche questo è un aspetto del tutto ragionevole: limitare questa dinamicità implica che il modello che si sta simulando sia sufficientemente statico. Questa limitazione fa sì che il paradigma PDES, allo stato attuale, non possa essere utilizzato per simulazioni di dinamiche complesse, che sono invece proprie di ambiti scientifici importanti come la biologia, la medicina, l'economia, ...

Nei capitoli successivi, dopo una presentazione dell'architettura di riferimento nella quale si opererà, verranno trattati gli aspetti progettuali ed implementativi necessari ad introdurre il concetto di *autonomicità di un sistema di gestione della memoria dinamica*.

Capitolo 2

Ambiente ROOT-Sim

Tutte le metodologie e le tecniche risultanti da questo lavoro sono state integrate all'interno di *ROme OpTimistic Simulator* (ROOT-Sim), una piattaforma open source di tipo PDES, sviluppata con tecnologia `C`, per la simulazione ottimistica general purpose. Questo capitolo ha come scopo quello di presentare l'architettura del kernel di simulazione di questa piattaforma di riferimento, al fine di mostrare meglio le motivazioni che sono alla base di alcune scelte progettuali ed implementative.

2.1 Livelli del Simulatore

In figura 2.1 viene mostrato lo schema dell'architettura di ROOT-Sim. Come si può vedere, essa prevede una stratificazione su più livelli, ognuno dei quali ha il compito di svolgere dei compiti particolari:

- **livello applicativo**: è lo strato al cui interno risiede il programma di simulazione creato dall'utente;
- **livello del kernel di simulazione**: è uno strato intermedio all'interno del quale risiede la piattaforma di simulazione, composta da tutti i suoi sottosistemi;
- **livello MPI**: è il livello costituito dalla libreria MPI (Message Passing Interface), utilizzata dalla piattaforma per rendere distribuita la simulazione, permettendo lo scambio di messaggi tra i vari LP.

Ciascun livello espone ai livelli superiori (tramite alcune API ad hoc) delle funzionalità e, al tempo stesso, usufruisce delle funzionalità dei livelli

inferiori. Per rendere molto semplice l'aggancio reciproco dei livelli, nonostante la complessità delle operazioni svolte da ciascuno di essi, le API esposte sono estremamente semplici.

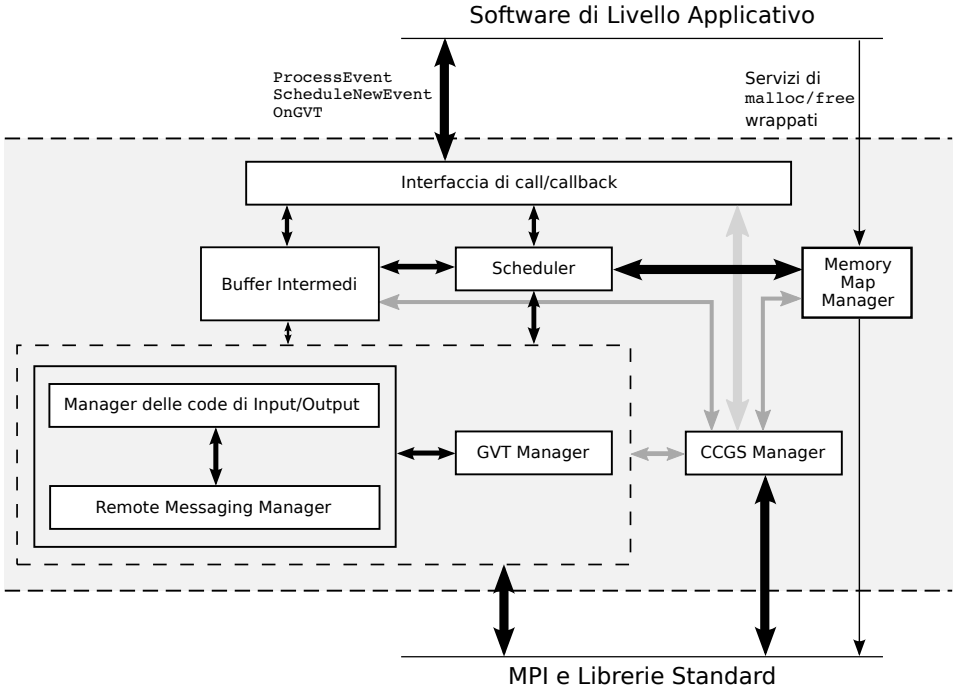

Figura 2.1: Schema dell'architettura di ROOT-Sim

2.1.1 Livello Applicativo

Il livello applicativo è quello strato software all'interno del quale si colloca l'applicazione ideata dal programmatore, concepita per simulare un modello.

Un *modello* è una rappresentazione concettuale — spesso una semplificazione, seguendo il principio del *Rasoio di Occam*[1] — del mondo reale o di una sua parte, capace di spiegarne il suo funzionamento, e che corrisponde all'oggetto modellato poiché è in grado di riprodurne alcune caratteristiche o comportamenti fondamentali.

[1]Tale principio, alla base del pensiero scientifico moderno, nella sua forma più immediata suggerisce l'inutilità di aggiungere nuove ipotesi ad una teoria, quando essa funziona.

Tipicamente, il livello applicativo è costituito da una porzione di codice che implementa un modello matematico, all'interno del quale gli eventi vengono utilizzati per trasportare informazioni tra un nodo e l'altro (o anche da un nodo a sé stesso) su come modificare/far evolvere la computazione di questo modello.

In ROOT-Sim, il livello applicativo comunica con il livello del kernel attraverso l'utilizzo di tre funzioni:

- `ScheduleNewEvent()`: permette di comunicare al kernel di simulazione un nuovo evento appena generato. Questo evento dovrà essere spedito dal kernel al suo corretto destinatario che si occuperà di processarlo secondo il suo scheduling;
- `ProcessEvent()`: permette al kernel della piattaforma di consegnare a ciascun LP il successivo evento da eseguire. L'ordine di causalità viene pertanto gestito dal kernel della piattaforma. Il processamento dell'evento viene discriminato in base alla tipologia cui esso appartiene, utilizzando il codice definito dallo stesso programmatore, basato su un costrutto di *multiway branching*;
- `OnGVT()`: tramite questa funzione la piattaforma comunica all'applicazione che è stato raggiunto uno stato *committed* (ossia non più annullabile) per tutti gli LP, pertanto è possibile verificare, tramite l'analisi di alcuni predicati, se la simulazione ha raggiunto la sua configurazione finale di terminazione.

Il callback `ProcessEvent()` accetta un insieme di parametri tramite i quali (i) è possibile identificare l'evento da processare (nella forma di una struttura dati definita dal programmatore dell'applicazione), (ii) identificare il processo logico che deve eseguire l'evento (nella forma di un codice numerico, corrispondente all'indice globale dell'LP) e (iii) specificare l'indirizzo di base dello stato associato all'LP che sta per eseguire l'evento. Inoltre, tramite un flag, consente al codice di livello applicativo — se siamo in presenza della prima chiamata a `ProcessEvent()` dell'intera simulazione — di eseguire delle routine di inizializzazione.

`ProcessEvent()` è implementato in tecnologia `C` con la possibilità di utilizzare i servizi di allocazione dinamica della memoria malloc e free, grazie all'integrazione di una libreria di gestione della memoria dinamica. Que-

sta libreria, presentata in due versioni (DyMeLoR, [TQ08], e Di-DyMeLoR, [PVQ09]), verrà presentata nei suoi dettagli nel paragrafo 2.2.7.

Il servizio `ScheduleNewEvent()` offerto dalla piattaforma può essere invocato dal codice di livello applicativo durante il processamento di un evento, al fine di iniettare nel sistema nuovi eventi. Per utilizzare questo servizio è necessario specificare unicamente il contenuto del nuovo evento ed il codice numerico per l'identificazione dell'LP destinatario.

`OnGVT()` è una funzione che viene invocata dal kernel di simulazione. Essa, inserita dal programmatore all'interno del livello applicativo, implementa quella parte del modello che consente di valutare un predicato globale di terminazione della simulazione. Dato lo snapshot dello stato globale S del sistema (identificato al momento del calcolo del GVT), l'applicazione (nella forma dell'i-simo processo logico) decide, analizzando $S_i \subseteq S$, se si può terminare o meno la simulazione.

2.1.2 Livello Kernel

Il livello del kernel è il nucleo centrale dell'architettura, in cui risiede la singola istanza della piattaforma di simulazione. Dal momento che la piattaforma è sia parallela che distribuita, l'ambiente di simulazione è ripartito in un certo numero di kernel, ciascuno dei quali ospita al proprio interno un certo numero di LP, assegnato in modo esclusivo ed equilibrato, per bilanciare il carico di lavoro.

Uno di questi kernel assume il ruolo di *master*: esso svolge il ruolo di coordinatore tra i kernel (gli altri sono chiamati *slave*) in tutte quelle operazioni che richiedono che qualcuno prenda una decisione.

Ciascuna istanza del kernel è organizzata in una serie di sottosistemi, ciascuno dei quali si preoccupa di svolgere determinate mansioni indispensabili all'esecuzione di una corretta simulazione. Questi moduli, interallacciati tra loro tramite alcune opportune interfacce interne non visibili al livello applicativo, verranno analizzati nel dettaglio nel paragrafo 2.2.

2.1.3 Livello MPI

Il kernel di simulazione di ROOT-Sim si appoggia, per lo scambio dei messaggi, alla libreria MPI. Essa viene utilizzata per la realizzazione di un ambiente di lavoro distribuito in cui gli LP che appartengono a differenti istanze del kernel di simulazione — poste sulla stessa macchina, o su nodi differenti della rete — scambiano tra di loro messaggi.

L'implementazione della logica MPI utilizzata in questa piattaforma è fornita dalla libreria open source OpenMPI, che fornisce al kernel di simulazione tutte le API per accedere ai servizi necessari all'invio ed alla ricezione dei messaggi stessi.

2.2 Sottosistemi di Controllo e Gestione

Come anticipato nel paragrafo 2.1.2 e mostrato in figura 2.1, il kernel di simulazione di ROOT-Sim è organizzato in sottosistemi. Essi, completamente disaccoppiati l'uno dall'altro, espongono ed utilizzano dei servizi interni per interallacciare le loro operazioni. Nei paragrafi successivi questi sottosistemi, parte vitale della piattaforma di simulazione, verranno analizzati nel dettaglio.

2.2.1 Sottosistema di gestione degli eventi

Il *sottosistema di gestione degli eventi*, che prende il nome di `queue_mgnt`, si preoccupa del controllo degli eventi associati a ciascun LP e dei metadati ad essi associati. Gli eventi sono organizzati in una serie di code doppiamente concatenate, all'interno delle quali vengono rispettati i vincoli di causalità.

A ciascun evento, infatti, sono associati due valori di *tempo logico* che ne definiscono l'istante in cui esso è stato generato (T_{gen}) e l'istante in cui deve essere processato (T_{exe}). Il sottosistema `queue_mgnt` si occupa dell'aggiornamento e della gestione degli accessi alle code, rispettandone l'ordine temporale[2] stabilito dai timestamp degli eventi.

Qualora il gestore degli eventi riceva un messaggio straggler, esso viene segnalato al sistema. In risposta a questo evento, la piattaforma avvia la procedura di rollback.

[2] Gli eventi ricevuti da un LP sono contenuti all'interno di messaggi che non arrivano necessariamente in ordine di timestamp, dal momento che la rete sottostante non garantisce alcun ordinamento.

2.2.2 Sottosistema per il GVT

Come presentato in [Fuj90, Jef85], i sistemi di simulazione ottimistici ad eventi discreti si basano su tecniche di salvataggio dello stato che permettono di effettuare operazioni di rollback ogni qual volta si verifichi una violazione della causalità. I log degli stati sono utilizzati esclusivamente per motivi di sincronizzazione, dal momento che vengono eliminati quando un nuovo valore del Global Virtual Time indica che essi si riferiscono ad una porzione della computazione che non può essere annullata (*committed*).

Il *sottosistema per il GVT*, denominato `gvt_mgnt`, si occupa di calcolare periodicamente il valore del Global Virtual Time, secondo un protocollo master-slave distribuito, tra tutti i kernel di simulazione dell'intera piattaforma.

Periodicamente il kernel master invia a tutti gli slave un messaggio di notifica. Questo fa sì che gli slave, in risposta, inviino un valore di timestamp pari al minimo locale tra i timestamp degli eventi in attesa di processamento per tutti gli LP. Il master, una volta ricevute tutte le risposte, calcola il minore tra questi valori (il GVT) e lo comunica nuovamente alle istanze del kernel, che si preoccuperanno di adottarlo.

Alla ricezione del nuovo GVT, inoltre, tutti i kernel di simulazione potranno effettuare una potatura delle strutture dati: tutte quelle etichettate con un timestamp inferiore al valore del GVT verranno considerate obsolete e rilasciate.

2.2.3 CCGS Manager

La piattaforma utilizza il calcolo del GVT anche per avviare la procedura di controllo della terminazione. Se in alcuni contesti è sufficiente controllare che un determinato lasso di tempo simulato è già trascorso, al fine di decidere di terminare la computazione, in scenari di simulazione generali è necessario implementare questo controllo come un predicato globale da valutare su uno stato globale coerente e non più annullabile (*CCGS*, Committed and Consistend Global State) (cfr. [Mat93]).

Il calcolo del GVT è una forma di predicato globale, chiamato *distributed infimum approximation* in [Tel91], che si basa, tra le altre cose, su informazioni associate ai messaggi in transito. Tuttavia, questo predicato considera soltanto i valori del tempo logico e non prende in considerazione informazioni sullo stato degli LP, informazioni che potrebbero giocare un ruolo rilevante

per decidere se far terminare o meno la simulazione, in contesti applicativi specifici.

ROOT-Sim offre un meccanismo efficiente per identificare e costruire un CCGS formato da una collezione di valori di stato (uno preso da ogni processo logico). Il meccanismo è efficiente sotto due aspetti differenti:

- non impone alcuna forma di coordinazione tra le attività di salvataggio dello stato di LP differenti, garantendo a ciascuno di essi una completa autonomia. Lo schema utilizzabile per catturare lo stato può essere uno qualsiasi di quelli presentati nel paragrafo 1.2.

- si basa su una politica di aggiornamento degli stati non più annullabili (supportata da un approccio simile al coasting forward) che si fonda su un metodo euristico che sfrutta unicamente le informazioni locali disponibili presso un singolo LP.

Le informazioni locali vengono utilizzate per determinare quando la fase di aggiornamento del checkpoint non più annullabile (*committed*) può terminare, assicurando allo stesso tempo l'assenza di mutue dipendenze tra gli stati degli LP che formano lo stato globale.

Il metodo euristico per l'identificazione e costruzione distribuita del CCGS è implementata in modo completamente trasparente rispetto al software applicativo. Quando lo stato di un LP appartenente allo stato globale deve essere ricostruito, il CCGS Manager salva lo stato corrente dell'LP in un'area temporanea. In seguito, ripristina l'ultimo checkpoint non più annullabile (*committed*) dell'LP, che viene in seguito aggiornato secondo lo schema euristico: vengono effettuate invocazioni successive di `ProcessEvent()`, al quale viene passato come indirizzo di base quello dello stato ripristinato. Durante questa procedura, tutti i messaggi in uscita prodotti dall'applicazione vengono bloccati, grazie a delle caratteristiche dei *Buffer Intermedi*.

Il CCGS Manager invoca per ogni LP la funzione `OnGVT()` per valutare un predicato che indichi se per un particolare LP la computazione possa terminare o meno, restituendo un valore booleano. L'istanza master del kernel di simulazione raccoglie tutti quanti questi risultati dopo che il GVT è stato calcolato. La computazione può terminare quando tutti i flag raccolti hanno valore `true`.

2.2.4 Sottosistema per la gestione degli stati

Il *sottosistema per la gestione degli stati* (che prende il nome di `state_mgnt`) si occupa di gestire le operazioni di salvataggio e di ripristino (in caso di roll-back) degli stati di ciascun LP, preservando la correttezza della simulazione.

ROOT-Sim è configurabile in modo tale che possa essere utilizzato un buon numero di tecniche di salvataggio degli stati, come presentato nel paragrafo 1.2.

2.2.5 Sottosistema per lo scheduling

Il *sottosistema per lo scheduling* si occupa della gestione della sequenzialità degli eventi di ciascun LP. Stabilisce l'ordine con cui si dovranno processare gli eventi, così da mantenere il più possibile inalterata la loro causalità, nel rispetto dell'ordine temporale.

Lo scheduler si occupa di selezionare l'LP (tra quelli gestiti dal kernel) che deve effettuare un passo di simulazione. ROOT-Sim mette a disposizione vari scheduler degli eventi. Secondo la configurazione di base, gli LP vengono selezionati con un algoritmo *STF* (Shortest Timestamp First): il gestore valuta qual è l'LP cui fa riferimento l'evento con il timestamp più basso e lo attiva.

2.2.6 Sottosistema per lo scambio dei messaggi

La generazione di un evento provoca l'invio di un messaggio all'LP destinatario (che potrebbe coincidere con il mittente). In un tale contesto è necessario fornire un sistema di spedizione e ricezione di messaggi che coinvolga coppie di LP.

Tutti gli eventi generati dall'applicazione vengono inseriti in un buffer circolare inizialmente vuoto. Successivamente il sottosistema per lo scambio dei messaggi analizza il buffer e consegna ai destinatari i messaggi.

Quando il kernel riceve in maniera asincrona i messaggi, li inserisce nelle code tramite il gestore degli eventi.

2.2.7 Gestore della Memoria Dinamica

Il *gestore della memoria dinamica* è forse la parte più complessa e delicata dell'intera piattaforma ROOT-Sim. Esso si preoccupa di consentire all'u-

tente l'utilizzo dei servizi standard di allocazione e deallocazione della memoria dinamica, tenendo traccia pertanto delle evoluzioni dello stato di ciascun LP nel tempo e nello spazio, garantendo che, in seguito ad un'operazione di rollback, esso verrà ripristinato ad uno stato coerente.

Nel tempo, in ROOT-Sim sono apparse due versioni differenti di gestori della memoria. Il primo, DyMeLoR, presentato in [TQ08], è stato il primo sottosistema in grado di gestire la memoria dinamica e stati a dimensione variabile nel tempo. Il secondo, Di-DyMeLoR, presentato in [PVQ09], è una naturale evoluzione del primo che permette alla piattaforma di effettuare checkpointing incrementale. Nei paragrafi successivi verranno descritti nel dettaglio entrambi i sottosistemi, poiché essi sono fondamentali per la creazione di un sottosistema autonomico di gestione della memoria, come vedremo nel capitolo 3.

DyMeLoR

La libreria DyMeLoR (Dynamic Memory Logger and Restorer), presentata in [TQ08], può essere considerata, da un punto di vista architetturale, come un wrapper dei servizi `malloc/free` dell'ANSI-C che viene frapposto, in maniera del tutto trasparente al programmatore, tramite delle semplici direttive a tempo di linking, tra il codice di livello applicativo e la libreria malloc tradizionale. Una schematizzazione di questo approccio è rappresentata in figura 2.2.

Come già precedentemente mostrato in figura 2.1, DyMeLoR offre un'API per l'integrazione con il kernel di simulazione, che consiste in un insieme di servizi che supportano operazioni di gestione della memoria orientate specificatamente al salvataggio ed al ripristino degli stati.

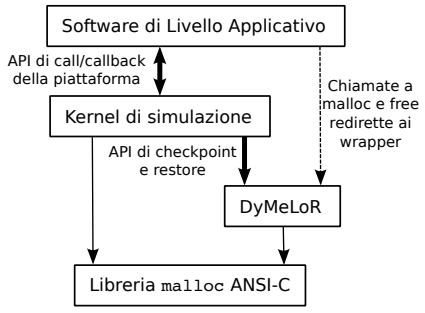

Figura 2.2: Architettura di DyMeLoR

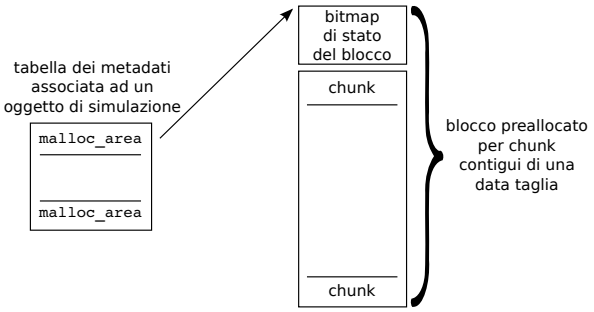

Figura 2.3: Strutture dati di DyMeLoR

DyMeLoR mantiene, per ciascun oggetto ospitato dal kernel di simulazione, una tabella di metadati (le cui entry sono chiamate `malloc_area`) come mostrato in figura 2.3. Ciascun elemento della tabella mantiene informazioni riguardanti un blocco di chunk contigui in memoria (come, ad esempio, la posizione in memoria del blocco), eventualmente allocata per servire richieste di memoria per quell'oggetto. Elementi differenti sono utilizzati per gestire chunk di dimensioni differenti.

Nel momento in cui viene ricevuta una richiesta per un chunk di una determinata dimensione, il blocco corrispondente viene allocato da DyMeLoR tramite una chiamata al vero servizio `malloc`. In pratica, in questo modo viene preallocato un certo numero contiguo di blocchi della stessa taglia, pronti per servire richieste future.

Questa preallocazione permette a DyMeLoR di utilizzare metadati molto concisi per l'identificazione dello stato di ciascun chunk (se *occupato* o *libero*) all'interno di un blocco. In particolare, viene utilizzata una bitmap di cosiddetti *status bit*. Per ottimizzare ancora di più l'utilizzo di memoria, essa viene posta in cima al blocco dei chunk, e viene creata solamente qualora il blocco in questione venga realmente allocato.

Qualora la tabella delle `malloc_area` arrivi a saturazione, essa può essere espansa nel caso in cui l'oggetto di simulazione faccia richiesta di nuovi chunk.

Le operazioni di log e restore in DyMeLoR sono eseguite con semplici tecniche di impacchettamento e spacchettamento dei dati. In un'operazione di salvataggio dello stato, i chunk correntemente in uso vengono impacchettati in un buffer contiguo (allocato dinamicamente tramite la sottostante

`malloc`), insieme alle entry di `malloc_area` attive e le bitmap di stato.

In un'operazione di ripristino dello stato, le strutture dati di un log vengono estratte dal buffer contiguo e rimesse al loro posto. Per far sì che le operazioni di deallocazione possano essere reversibili, la `malloc_area` mantiene anche le informazioni relative al tempo logico (se disponibili) in cui i chunk all'interno di un dato blocco sono stati tutti rilasciati. Un blocco con tutti i chunk non allocati ed il cui ultimo rilascio sia avvenuto prima del GVT può essere deallocato tramite una chiamata a `free` verso la libreria malloc sottostante. In quel caso, la corrispondente `malloc_area` viene impostata a non attiva.

L'allocazione dei chunk all'interno di ciascun blocco avviene in maniera simile all'algoritmo di Linux per la selezione del successivo descrittore di file da assegnare, quando si apre un canale di I/O. Questo algoritmo mantiene bassa la frammentazione e tende a mantenere raggruppati i chunk assegnati in cima al blocco di memoria, con una conseguente diminuzione della latenza delle operazioni di log, causata da un'esaltata località.

Di-DyMeLoR

Come si può intuire, Di-DyMeLoR (Dirty Dynamic Memory Logger and Restorer) è un'evoluzione di DyMeLoR basata sull'intercettazione delle scritture e sull'instrumentazione del codice. Questo gestore della memoria si avvale di un tool di instrumentazione simile a quello presentato in [WP96] e descritto nel paragrafo 1.2.3, studiato però per operare su architetture *Intel IA-32* ed *x86-64*.

Tramite l'utilizzo di nuove strutture dati e moduli, questa libreria è in grado di utilizzare le informazioni relative alle operazioni di scrittura sulla memoria dell'applicazione per tracciare le attività di aggiornamento. In questo modo è possibile migliorare l'efficienza del sistema, tramite l'utilizzo di un approccio incrementale.

In Di-DyMeLoR le strutture dati originali per la gestione della mappa di memoria sono state espanse per potersi occupare in maniera esplicita della costruzione di log degli stati completi a partire dal salvataggio incrementale di quelle sole aree che sono state sporcate dall'ultima operazione di log.

Per tracciare i chunk sporcati è stata associata a ciascun blocco di chunk una seconda bitmap di cosiddetti *dirty bit*. Come si può osservare in figura 2.4, la bitmap è situata all'interno della stessa area di memoria puntata

dalla `malloc_area` corrispondente, che contiene anche l'originale use bitmap. Questa eredita le stesse caratteristiche della bitmap di stato originale.

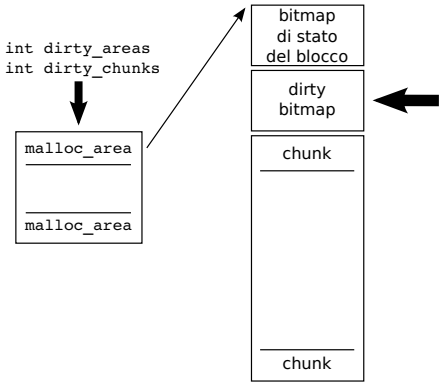

Figura 2.4: Strutture dati di Di-DyMeLoR

Pertanto, l'occupazione aggiuntiva di memoria necessaria a determinare quali chunk siano stati sporcati dall'ultima operazione di log è ben scalata rispetto alla dimensione della memoria assegnata all'applicazione.

Per le operazioni di salvataggio e ripristino dello stato, è necessario anche tracciare quali tra i metadati abbiano subito delle modifiche: per questo motivo all'interno della struttura `malloc_area` sono stati aggiunti i seguenti campi di tipo intero (come si può vedere dalla figura 2.4):

- `dirty_area`: viene utilizzato come flag per indicare se è avvenuta un'operazione di qualsiasi tipo (tra allocazione, deallocazione, o scrittura di un chunk) all'interno dell'area, dall'ultima operazione di log;
- `dirty_chunks`: tiene il conto del numero di chunk in uso che sono stati sporcati, all'interno dell'area, dall'ultima operazione di log.

Conformemente al modello di DyMeLoR originale, nel caso in cui l'indirizzo interessato da una scrittura risieda all'esterno della mappa di memoria dell'oggetto di simulazione correntemente in esecuzione (ad esempio, riferimenti a variabili globali esterne all'heap), il gestore della memoria restituisce il controllo.

Nei paragrafi seguenti, analizzerò più nel dettaglio le scelte progettuali ed implementative alla base di Di-DyMeLoR.

Tool di Instrumentazione Il tool di instrumentazione è il nucleo centrale dell'architettura di Di-DyMeLoR. Esso è stato sviluppato per operare su architetture *IA-32* ed *x86-64*. Caratteristica principale di questo *Instruction Set* è la lunghezza variabile[3]. Essa era molto utile negli anni '70 ed '80, poiché permetteva di risparmiare molta memoria, allora estremamente costosa.

La versione a 32 bit di questa architettura, ben descritta in [Inta, Intb, Intc], prevede 8 registri a 32 bit *general-purpose*[4] associati a dei codici numerici da 0 ad 8.

Sono inoltre previsti sei *segment-registers* da 16 bit, chiamati `CS`, `DS`, `SS`, `ES`, `FS`, `GS`, che però non sono di interesse qualora si utilizzi un sistema operativo `Unix-like`.

Il formato delle istruzioni dell'IA-32 prevede vari campi, alcuni dei quali possono essere o meno presenti, come viene rappresentato in figura 2.5. Ogni singolo campo delle istruzioni ha un significato preciso. Nella tabella 2.1 viene riportato in sintesi il loro uso. Per una trattazione più approfondita, rimando a [Pel07].

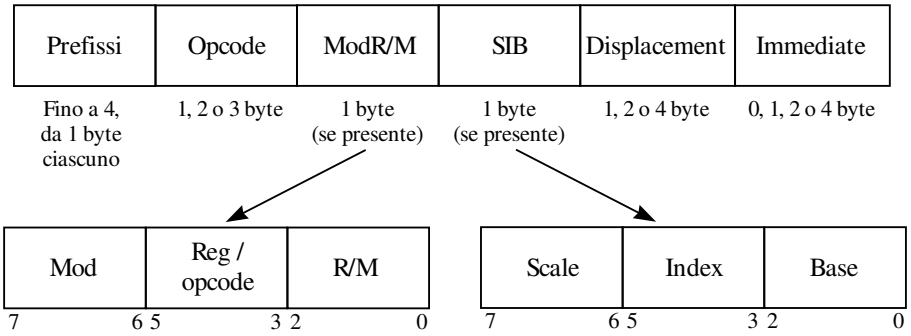

Figura 2.5: Schema del formato delle istruzioni per l'IA-32

Il formato istruzioni della versione a 64 bit è sostanzialmente identico al corrispettivo a 32 bit. La differenza più significativa risiede nel fatto che è stato introdotto un nuovo prefisso, denominato `REX`, posizionato tra i 4 prefissi originali e l'opcode principale.

Il prefisso `REX`, come mostrato in figura 2.6, ha i primi 4 bit preimpostati a `0100`[5]. I restanti quattro bit del prefisso permettono di estende-

[3]Il limite imposto è comunque di 16 byte.

[4]I registri sono `eax`, `ecx`, `edx`, `ebx`, `esp`, `ebp`, `esi` ed `edi`.

[5]Nell'architettura IA-32 i prefissi `REX` corrispondono ad un insieme di 16 opcode, che

Prefissi	I prefissi specificano dettagli sul comportamento delle istruzioni. Permettono di ripeterle fino al verificarsi di una condizione, di dare informazioni sull'esito di salti condizionali, di modificare le dimensioni dei dati coinvolti rispetto a quanto specificato dall'opcode.
Opcode	Di formato variabile. Permette di identificare l'operazione da eseguire. Il primo byte identifica una *famiglia di istruzioni*.
ModR/M	Se l'istruzione punta un operando in memoria, questo byte specifica qual è la forma di indirizzamento.
SIB	Alcune codifiche del byte ModR/M specificano la presenza di un ulteriore byte per l'indirizzamento, che consente di identificare un indirizzo in memoria scomponendolo in *Scala*, *Indice* e *Base*.
Displacement	Alcune modalità di indirizzamento si servono di uno *spiazzamento*, che può avere una lunghezza di 1, 2 o 4 byte, a seconda dell'istruzione che lo utilizza.
Immediate	Se un'istruzione specifica un *dato immediato*, ossia una quantità di informazioni cablate nell'istruzione, esso viene collocato sempre in coda all'istruzione stessa.

Tabella 2.1: Significato dei campi dell'Instruction Set x86

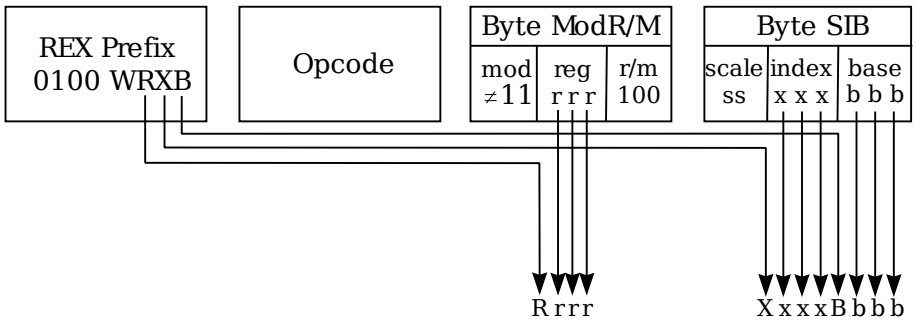

Figura 2.6: Prefisso REX

re l'architettura a 32 bit specificando operandi a 64 bit (dati, indirizzi o registri).

Un insieme importante di istruzioni è costituito da quelle che permettono di lavorare sulle stringhe[6], di cui le più interessanti per questo lavoro sono quelle che consentono la scrittura o la copia su aree di memoria di dimensione arbitraria: `stos` e `movs`. La prima permette di scrivere un certo numero di ripetizioni del valore contenuto nel registro `AX`. La seconda, invece, consente di copiare un'area di memoria in un'altra, della stessa dimensione. Appare pertanto evidente che il tool di instrumentazione dovrà essere in grado di riconoscere ed interpretare correttamente anche queste istruzioni, poiché esse possono essere utilizzate per effettuare un aggiornamento dello stato di un LP.

Per identificare un'area di memoria, l'architettura *IA-32* (ed *x86-64* fornisce una gran quantità di modalità di indirizzamento, mostrate in figura 2.7. Un operando in memoria viene identificato specificando, al più, 5 variabili: un *registro di segmento*, un *indirizzo di base*, un *valore di indice*, una *scala*[7] ed uno *spiazzamento*. L'indirizzo di base ed il valore di indice sono sempre memorizzati all'interno di registri, scala e spiazzamento sono invece cablati all'interno dell'istruzione.

Nella versione a 64 bit dell'architettura vi è una forma di indirizzamento aggiuntiva, chiamata *RIP-Relative Addressing*.

Se nell'architettura a 32 bit un indirizzamento relativo all'*instruction pointer* (o *program counter*) è possibile soltanto con le istruzioni di trasferimento del controllo, nell'architettura a 64 bit le istruzioni che utilizzano il byte ModR/M possono indirizzare in maniera relativa al valore corrente del registro `RIP`.

---

occupano un'intera riga della mappa degli opcode. In particolare, occupano le posizioni `0x40` ÷ `0x4f`.

Questi opcode (a byte singolo) corrispondono a delle istruzioni valide (`inc` e `dec`). Nell'architettura a 64 bit, pertanto, queste istruzioni non sono più disponibili, ma se ne possono usare alcune equivalenti a 2 byte, inizianti con l'opcode di escape `0xff`.

[6] Per *stringa* si intende, secondo la definizione classica, una sequenza ordinata di simboli. In questo caso particolare i simboli sono costituiti dai valori assunti da singoli byte.

[7] Appare subito evidente che la complessità di questa modalità di indirizzamento è stata concepita per identificare in maniera concisa, utilizzando codice macchina efficiente, dati appartenenti a strutture dati non primitive, quali array o `struct`.

$$\left\{\begin{array}{c}\text{CS:}\\ \text{DS:}\\ \text{SS:}\\ \text{ES:}\\ \text{FS:}\\ \text{GS:}\end{array}\right\}\left[\left\{\begin{array}{c}\text{EAX}\\ \text{EBX}\\ \text{ECX}\\ \text{EDX}\\ \text{ESP}\\ \text{EBP}\\ \text{ESI}\\ \text{EDI}\end{array}\right\}\right] + \left[\left\{\begin{array}{c}\text{EAX}\\ \text{EBX}\\ \text{ECX}\\ \text{EDX}\\ \text{EBP}\\ \text{ESI}\\ \text{EDI}\end{array}\right\} * \left\{\begin{array}{c}1\\ 2\\ 4\\ 8\end{array}\right\}\right] + [displacement]$$

Figura 2.7: Metodo di indirizzamento in memoria per l'IA-32

L'instrumentatore è, di fatto, un interprete dell'instruction set Intel: poiché il formato delle istruzioni è variabile, il tool legge il codice macchina un byte alla volta, eseguendo il *fetch* dell'istruzione. Ogni volta che viene identificata un'istruzione che può modificare una locazione di memoria, viene anteposta ad essa una `call` ad un modulo chiamato `monitor`. Allo stesso tempo, in una tabella, vengono memorizzate varie informazioni circa l'istruzione appena decodificata, secondo lo schema riportato in figura 2.8.

```
struct _insn {
  char *instruction;
  enum addr_method addr_method[3];
  enum operand_type operand_type[3];
  void (*esc_function)(struct disassembly_state *);
  bool to_be_instrumented;
};
```

Figura 2.8: Struttura delle righe della tabella di istruzioni

Tramite questa tabella, pertanto, con pochi passaggi è possibile ricostruire l'esatta destinazione e l'esatta taglia della scrittura che l'istruzione in questione andrà a compiere. Per una trattazione completa del funzionamento di questo modulo rimando nuovamente a [Pel07].

Tracciamento delle scritture Tramite un'API interna chiamata `dirty_mem()`, il modulo `monitor` comunica a Di-DyMeLoR qual è la porzione dello stato che è stata modificata dall'istruzione di aggiornamento della memoria che ha causato l'attivazione della routine di monitoraggio.

Il gestore della mappa di memoria, così informato dell'aggiornamento, andrà a scandire la tabella delle `malloc_area` dell'LP correntemente in esecuzione. Qualora l'indirizzo notificato alla funzione `dirty_mem` sia afferente ad una qualche area dell'LP corrente, il gestore della mappa di memoria calcola quale (o quali) chunk contengono la regione aggiornata. A questo punto, i bit della dirty bit relativi ai chunk interessati vengono impostati ad 1.

Qualora l'indirizzo notificato non appartenga a nessuna area della mappa di memoria dell'LP corrente, il modulo `dirty_mem` non modifica alcuna delle strutture dati.

Operazioni di salvataggio degli stati Le attività di salvataggio degli stati di Di-DyMeLoR sono state differenziate tra *log completi* e *log incrementali*. Entrambi i tipi di log consistono in una serie di operazioni di impacchettamento di informazioni all'interno di un buffer contiguo in memoria, allocato tramite una chiamata a `malloc`. Tuttavia vengono impacchettate informazioni differenti.

Un'operazione di **log completo** coincide con l'operazione di log originariamente supportata da DyMeLoR. In essa, pertanto, non vengono salvate le dirty bitmap. L'unica differenza risiede proprio nella loro gestione: completato il log, infatti, le dirty bitmap vengono azzerate.

Un'operazione di **log incrementale** effettua, invece, operazioni di *impacchettamento* differenti a seconda del valore corrente delle strutture dati. Per ciascuna `malloc_area` attiva, si possono verificare i seguenti casi:

A: `dirty_area` vale 1 e `dirty_chunks` vale 0. In questo caso la `malloc_area` viene impacchettata nel buffer di log insieme alla bitmap di stato, indicando così l'allocazione dei chunk all'interno di un dato blocco. La dirty bitmap e i chunk, tuttavia, non vengono salvati;

B: `dirty_area` vale 1 e `dirty_chunks` è maggiore di 0. In questo caso la `malloc_area` viene impacchettata all'interno del buffer di log, insieme a tutti chunk correntemente assegnati che sono stati sporcati. Tutti gli altri chunk in uso non vengono salvati;

C: `dirty_area` vale 0. In questo caso, non viene memorizzata alcuna informazione circa la `malloc_area`.

Come nel caso dei log completi, i log incrementali causano un reset completo di tutte le strutture dati necessarie al tracciamento delle modifiche. Questo avviene indipendentemente da quale dei tre casi appena descritti si verifichi.

Voglio sottolineare, infine, che le operazioni di salvataggio incrementale degli stati non richiedono assolutamente di essere eseguite prima del processamento di ogni evento. Esse infatti si basano sul riconoscimento di porzioni di memoria sporcate dall'ultimo log, indipendentemente dal numero di eventi che causano le modifiche alla memoria. La ricostruzione dello stato si adatta quindi perfettamente sia alla modalità CSS che a quella SSS, descritte nei paragrafi 1.2.2 e 1.2.1

Operazioni di ripristino degli stati In modo simile a DyMeLoR, ciascun log viene etichettato con il tempo di simulazione corrente e tutti i log (sia completi, sia incrementali) vengono collegati tra loro in una catena.

Quando si presenta la necessità di eseguire un'operazione di ripristino verso un tempo di simulazione T, viene effettuata una ricerca all'interno della catena per determinare il log più recente con tempo minore o uguale di T.

Qualora il log trovato sia completo, viene eseguita un'operazione di ripristino analoga a quella compiuta originariamente da DyMeLoR. Qualora il log incontrato, invece, sia di tipo incrementale, entra in gioco un algoritmo differente. In particolare, vengono iterati i seguenti passi, all'indietro attraverso la catena di log, partendo da quello identificato come stato da ripristinare:

1. una `malloc_area` trovata all'interno di un buffer di log, che non sia ancora stata ripristinata, viene rimessa a posto all'interno della tabella dei metadati. Inoltre, viene ripristinata dal buffer anche anche la bitmap di stato[8];

2. ciascun chunk all'interno del buffer di log (associato alla `malloc_area` corrente) che non sia ancora stato ripristinato in un'iterazione precedente viene ricopiato al suo posto nel suo blocco di memoria.

[8]Si ricordi che, indipendentemente dal tipo di log e dal caso specifico di log incrementale, una `malloc_area` viene sempre associata alla sua bitmap di stato, per garantire la possibilità di ripristino di operazioni di allocazione e deallocazione.

La procedura iterativa di ripristino si ferma quando tutte le `malloc_area` sono state ripristinate insieme a tutti i chunk assegnati. Anche se, in principio, questo potrebbe richiedere un numero non definito di passi iterativi all'indietro lungo la catena di log, in pratica l'operazione di restore può essere finalizzata nel momento in cui viene incontrato un log completo durante l'attraversamento della catena. Infatti tutti i chunk in uso che non sono ancora stati ripristinati diventano immediatamente disponibili per la copia nel log completo.

Per ottimizzare l'individuazione dei chunk non ancora ripristinati, la procedura iterativa di ripristino si appoggia a delle bitmap temporanee (una per ciascuna `malloc_area`) su cui viene eseguita una coppia di veloci operazioni bit a bit di tipo OR-XOR, ogni volta che viene estratta da un buffer di log una dirty bitmap. Questo procedimento è illustrato in figura 2.9.

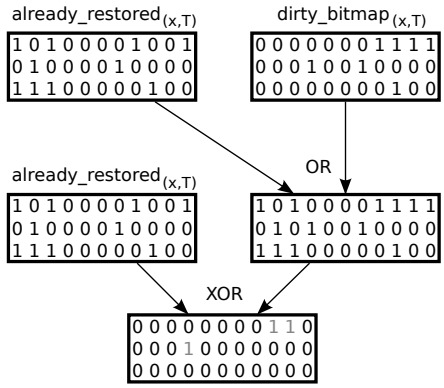

Figura 2.9: Operazione OR-XOR sulle bitmap

La bitmap temporanea $already\_restored_{(x,T)}$ è quella associata ad una `malloc_area` x e ad un log con timestamp T e mantiene le informazioni relative a quali chunk sono già stati ripristinati. La bitmap $dirty\_bitmap_{(x,T)}$ è invece una di quelle estratte da un buffer di log marcato con timestamp T ed afferente ad una `malloc_area` di indice x. Come si può vedere, l'operazione descritta in figura permette di identificare con due sole veloci operazioni quali siano i soli chunk presenti nel log in fase di processamento che dovranno essere ripristinati.

Caching dei riferimenti In DyMeLoR sono stati volutamente evitati degli header per i chunk, per evitare di dover effettuare log e restore di metadati troppo grandi. Pertanto, quando un chunk viene rilasciato, non è possibile utilizzare alcuna struttura dati per accedere in maniera rapida alla `malloc_area` coinvolta nell'operazione.

Per velocizzare queste operazioni DyMeLoR forniva un sottosistema di *direct-map caching* a livello software, implementato come una tabella hash, con le righe della tabella formate dalla coppia:

$$< chunk\_start\_address, m\_area\_index >$$

L'identificazione di una `malloc_area` a partire da un indirizzo di memoria diventa ancora più critica nella nuova versione Di-DyMeLoR: il gestore della mappa di memoria, infatti, deve recuperare la `malloc_area` per aggiornarne i metadati ogni volta che riceve una segnalazione da `update_tracker`, che è un evento molto più frequente di quello di una deallocazione.

Inoltre, in Di-DyMeLoR c'è la necessità di recuperare l'area corretta a partire da un indirizzo di memoria che non coincide necessariamente con il bordo superiore del chunk (come avviene invece per le operazioni di `free`), poiché `update_tracker` potrebbe catturare un'operazione di scrittura relativa ad una locazione di memoria posta a metà di un chunk. Per affrontare questa situazione la cache è stata estesa, portando le linee ad avere una forma rappresentata dalla tripla:

$$\langle chunk\_start\_address,\ chunk\_end\_address,\ m\_area\_index \rangle$$

La cache diventa pertanto in grado di accettare un input multiset. L'indirizzo iniziale di una scrittura intercettato da `update_tracker` viene privato di n bit meno significativi, dove n è scelto in modo tale da far collidere tutti gli indirizzi di uno stesso chunk nella stessa riga. In realtà, posto che la dimensione dei chunk forniti all'applicazione possa essere differente[9], n è stato scelto come valor medio tra il numero di bit necessari per generare una collisione tra i chunk più piccoli e quelli più grandi (tra quelli gestiti

[9]Ricordo che, così come avviene per la libreria `malloc`, Di-DyMeLoR gestisce dimensioni dei chunk che siano potenze di 2, con una dimensione massima parametrizzabile, generalmente impostata a 32KB.

da Di-DyMeLoR), moltiplicato per un fattore di scala teso a polarizzare il valore verso i chunk più piccoli[10].

Confronto con gli altri Sistemi di State Saving DyMeLoR e Di-DyMeLoR partono da idee condivise con altri sistemi di state saving, presentati nel paragrafo 1.2. Tuttavia, esso aggiunge ulteriori caratteristiche innovative.

In tabella 2.2 sono riassunte tutte le caratteristiche di DyMeLoR e Di-DyMeLoR in confronto con gli altri sistemi di salvataggio degli stati. Come si può notare, DyMeLoR si pone come un gestore della memoria che offre delle modalità di salvataggio dello stato estremamente semplici (che però possono essere espanse tramite l'utilizzo di politiche di selezione dell'intervallo di checkpointing analoghe a quelle degli altri lavori confrontati) ma che si concentra sull'offerta di servizi che consentano, in maniera del tutto trasparente all'utente, di gestire stati che — dinamicamente durante l'esecuzione della simulazione — possono crescere (o decrescere) all'interno della memoria dinamica. È questo il primo tentativo, in letteratura, di realizzazione di un approccio di questo tipo.

Di-DyMeLoR, invece, partendo da quanto proposto in DyMeLoR, fa un passo in avanti consentendo — sempre in maniera del tutto trasparente — la realizzazione di uno schema di salvataggio dello stato incrementale, mostrandosi così come il primo gestore della memoria capace di gestire l'incrementalità senza richiedere al programmatore di specificare quali siano le porzioni di memoria da monitorare (porzioni che, di fatto, non sono neppure note a priori, poiché lo stato può variare di dimensione).

[10] I chunk di dimensione medio-piccola sono, statisticamente, quelli più richiesti da un software di livello applicativo.

State Saving	Fossil Collection	Coasting Forward	Checkpointing Periodico	Periodo Adattativo	Stato Dinamico	Incrementalità	Granularità Arbitraria	Trasparenza	Instrumentazione	Granularità degli Eventi	Multiplexing	Modello Analitico	Misure Temporali	Euristiche	Autonomicità	Stabilità alle Fluttuazioni
[Jef85]																
[Jef90]	✓															
[Bel92]	✓	✓	✓													
[PW93]	✓	✓	✓	✓								✓	✓			
[RA94]	✓	✓	✓	✓								✓				
[SR96]	✓	✓	✓	✓						✓		✓				
[FW95]	✓	✓	✓	✓									✓	✓		
[Qua98]	✓	✓	✓	✓						✓		✓				
[BS93]	✓	✓				✓	✓									
[SQ05a]	✓	✓				✓		✓								
[RLAM96]	✓	✓				✓										
[WP96]	✓	✓	✓	✓		✓		✓	✓				✓			
[GUCF97]	✓	✓		✓		✓					✓					
[Qua01]	✓	✓	✓	✓						✓		✓				
DyMeLoR	✓	✓	✓	✓	✓			✓								
Di-DyMeLoR	✓	✓	✓	✓	✓	✓	✓	✓	✓				✓	✓		

Tabella 2.2: DyMeLoR e Di-DyMeLoR a confronto

Capitolo 3

Gestore per il Log/Restore Autonomico

Come mostrato nel paragrafo 1.1, l'approccio di sincronizzazione ottimistica (basata sul rollback per recuperare possibili violazioni nell'ordine dei timestamp, dovute all'assenza di politiche di arresto della simulazione fino al raggiungimento di uno stato sicuro) tende a favorire un aumento delle prestazioni, quale che sia l'architettura di simulazione o l'applicazione.

Tuttavia, il processo di progettazione e sviluppo di supporti ottimizzati per il salvataggio degli stati è uno dei maggiori ostacoli per la costruzione di sistemi di simulazione ottimistica efficienti. Questo processo diventa ancora più complesso quando si cerca di garantire una trasparenza completa nei confronti del livello applicativo.

In questo capitolo presenterò un'architettura progettata secondo il paradigma della computazione autonomica, che si preoccupa contemporaneamente di questioni di trasparenza e performance, pubblicato in [VPQ10].

Un *sistema autonomico* è un sistema che opera e serve il suo scopo gestendo se stesso, senza alcun intervento esterno, anche in caso di cambiamenti dell'ambiente circostante.

In figura 3.1 viene mostrato lo schema generale di un tale sistema. Parte fondamentale di esso è la capacità di percepire l'ambiente (le *Misure* M_i), capacità che permette di osservare il contesto operazionale esterno. Insita in un sistema autonomico è la conoscenza dell'*Obiettivo*, (scopo, o intenzione), e la *Conoscenza* di come operare su sé stesso (ad esempio l'avvio, la configurazione, l'interpretazione delle misure, ...) senza interventi esterni. Le vere e proprie operazioni del sistema autonomico sono guidate dalla *Logica*,

che è responsabile di prendere le giuste decisioni per raggiungere lo scopo, partendo dalle misurazioni del contesto operazionale.

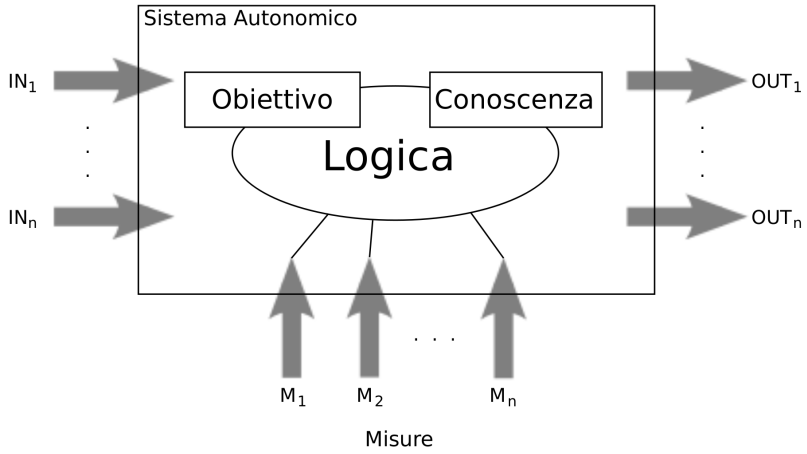

Figura 3.1: Modello concettuale del paradigma autonomico

Da questo modello si deduce che il funzionamento di un sistema autonomico è *guidato dall'obiettivo*. Ciò include la sua *missione* (ossia, il servizio che ci si attende venga offerto), le *politiche* (ossia, l'insieme dei comportamenti basilari) e l'"*istinto di sopravvivenza*". Se vediamo un sistema di questo tipo come un sistema di controllo, esso verrebbe realizzato come una funzione a feedback di errore o come un sistema assistito da euristiche.

Anche se lo scopo (e quindi il funzionamento) di un sistema autonomico varia da sistema a sistema, ognuno di essi deve esibire un insieme minimo di proprietà per raggiungere il suo obiettivo:

1. *Automazione*, ossia la capacità di auto controllare le proprie funzioni interne e le proprie operazioni. Per raggiungere questo scopo, un sistema autonomico deve essere *autocontenuto*, e deve essere in grado di avviarsi da solo, senza alcun intervento manuale o aiuto esterno. Inoltre, la conoscenza necessaria all'avvio del sistema deve essere implicita all'interno dello stesso.

2. *Adattatività*, che consente ad un sistema autonomico di modificare le proprie operazioni (ossia la propria configurazione, stato e funzioni). Questa proprietà consente al sistema di reagire a cambiamenti spaziali e temporali del proprio contesto operazionale, sia sul lungo termine (ot-

timizzazione ambientale), sia sul breve termine (condizioni eccezionali come attacchi malevoli, errori, ...).

3. *Consapevolezza*, ossia quella capacità di un sistema autonomico di monitorare (misurare) il proprio contesto operazionale così come il proprio stato interno, in modo tale da verificare se le proprie operazioni correnti siano le migliori per raggiungere il proprio scopo. La consapevolezza controlla l'adattività del comportamento operazionale in risposta a cambiamenti del contesto o dello stato.

Il modello di sistema autonomico qui proposto è coerente sia con la definizione di un *sistema di controllo*, sia con quella di un *agente di Intelligenza Artificiale*. Ciò è ragionevole, dal momento che, in pratica, un sistema di controllo ed un agente di Intelligenza Artificiale possono implementare un sistema autonomico, a patto che essi esibiscano le proprietà appena citate.

Il sistema che propongo in questo lavoro, come vedremo, rispetta il modello di autonomicità e migliora i sistemi di Memory Management presentati nel paragrafo 2.2.7 in termini di performance e trasparenza, dal momento che:

- Consente al programmatore di utilizzare costrutti standard per l'allocazione/deallocazione della memoria dinamica, permettendo che lo stato di un oggetto di simulazione sia sparpagliato su porzioni di memoria non contigue;
- Adotta alternativamente, secondo uno schema a fasi (e del tutto trasparentemente) le modalità di log e restore incrementale e non incrementale, scegliendo la modalità migliore attraverso delle misurazioni di parametri che consentono di descrivere le dinamiche dell'esecuzione, tramite l'utilizzo di un innovativo modello analitico;
- Esegue le operazioni di log e restore in modo altamente ottimizzato, attraverso l'adozione di schemi classici di ottimizzazione dei parametri che determinano l'overhead finale di ciascuna modalità d'esecuzione.

In questo capitolo discuterò i dettagli progettuali ed implementativi alla base della corretta realizzazione di un tale sistema.

3.1 Versioni Multiple del Codice Applicativo

Il blocco principale per la costruzione del sistema autonomico in questione, è il gestore della mappa di memoria presentato nel paragrafo 2.2.7, nelle due versioni, incrementale e non.

Il progetto dei due Memory Map manager è stato ampliato, così da fornire una coesistenza ottimizzata delle due modalità di log differenti. Una possibile soluzione a questo aspetto sarebbe quella di aggiungere (per ogni processo logico) un flag che indichi al modulo di tracciamento delle scritture se il sottosistema di gestione della memoria sta operando in modalità incrementale o meno. Nel secondo caso, il monitor non dovrebbe eseguire alcuna operazione per identificare il chunk di memoria da marcare come sporco, ma potrebbe semplicemente restituire il controllo al normale flusso d'esecuzione.

Tuttavia, se si adottasse questa soluzione, l'esecuzione della modalità di checkpointing non incrementale coinvolgerebbe l'esecuzione di porzioni di codice superflue: la chiamata alla funzione di monitoring, la preparazione della finestra di stack, il controllo di un flag, e la successiva restituzione del controllo al codice applicativo. In questo modo, si aggiungerebbe un overhead alla normale esecuzione e la modalità non incrementale non sarebbe ottimizzata.

L'approccio adottato è invece differente: tramite una modifica manuale degli eseguibili generati, partendo dallo stesso insieme di moduli di livello applicativo, vengono generate due sezioni .text all'interno del programma finale, una contenente una versione non instrumentata del codice applicativo, e l'altra contente la controparte instrumentata. Queste due sezioni vengono poste in modo del tutto trasparente all'interno di sezioni di memoria virtuale differenti, utilizzando delle caratteristiche standard del linker GNU `ld`. Allo stesso tempo però, tramite il tool di instrumentazione, le tabelle dei simboli di queste due versioni vengono modificate, così da mostrare al Kernel di simulazione le interfacce richieste, ossia le funzioni `ProcessEvent()`, `ScheduleNewEvent()` ed `OnGVT()`, attraverso dei simboli differenti.

Le sezioni .rodata, corrispondenti ai dati in sola lettura, vengono modificate così che possano occupare aree di memoria virtuale differenti, all'interno dell'eseguibile finale. Le sezioni .data e .BSS, contenenti i dati che l'applicazione può utilizzare e modificare durante la sua esecuzione, vengono sovrapposte all'interno della stessa area di memoria virtuale, così da forni-

re una singola copia dei dati inizializzati e non inizializzati, accessibili da entrambe le versioni del codice applicativo.

Uno schema di questo processo, che consente la coesistenza di due versioni differenti del codice applicativo all'interno dello stesso eseguibile, in maniera tale che i dati possano essere correttamente condivisi, è presentato in figura 3.2.

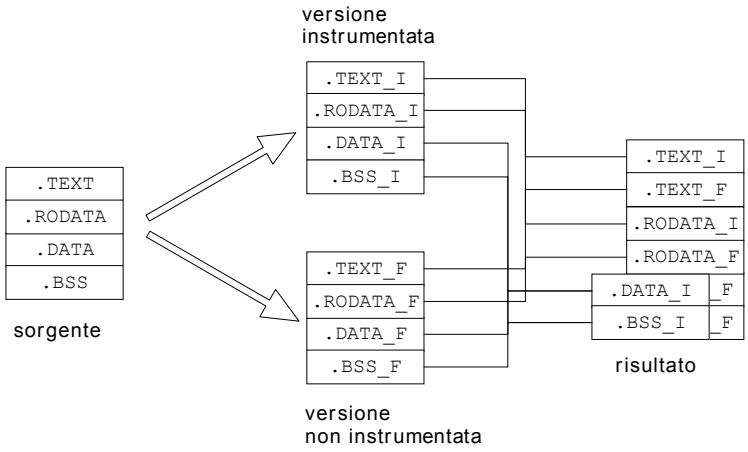

Figura 3.2: Generazione delle versioni multiple di codice

Una volta che l'eseguibile finale è generato ed eseguito, uno switch (a livello del kernel di simulazione) tra le due modalità di checkpointing si limita a riassegnare i puntatori alle funzioni di interfaccia tra il kernel stesso ed il codice di livello applicativo ai simboli associati alla versione del codice duplicato, che consente l'esecuzione nella modalità richiesta. Adottando questa soluzione, ciascuna modalità di checkpointing viene supportata secondo uno schema d'esecuzione ottimizzato in cui viene evitato qualsiasi tipo di overhead, mentre l'esecuzione degli eventi può essere portata avanti secondo la modalità di checkpointing richiesta.

Lo schema appena presentato richiede soltanto l'utilizzo di un numero maggiore di indirizzi virtuali, dal momento che vi è la presenza di due versioni dei moduli applicativi. Tuttavia, ciò non rappresenta un problema reale se si considera la tendenza dei produttori di macchine a spostarsi sui

processori a 64 bit, fattore che consente l'utilizzo di un amplissimo raggio di indirizzi virtuali di memoria. Inoltre, le sezioni testo degli eseguibili, tipicamente, occupano una porzione relativamente ristretta degli indirizzi virtuali disponibili.

3.2 Modelli di Valutazione di Costo

Dopo aver spiegato come si possono far coesistere le due versioni del codice applicativo, volte a supportare le modalità di checkpointing incrementale e non incrementale, presento qui i modelli che sono stati sviluppati per valutare l'overhead per evento dovuto alle operazioni di salvataggio e ripristino dello stato degli oggetti di simulazione.

Questi modelli hanno come base il lavoro in [RA94] per il salvataggio periodico non incrementale, al quale vengono aggiunte (i) una porzione di equazione per catturare i meccanismi interni propri del gestore della mappa di memoria (ossia il costo della gestione dei metadati che identificano i layout di memoria sparsa) e (ii) un'estensione che gestisce il caso del checkpointing incrementale dell'architettura presentata. Il modello in [RA94] descrive l'overhead del log/restore con una granularità all'oggetto di simulazione. Questa caratteristica viene ereditata nel mio modello, fornendo così uno schema autonomico che consente di ottimizzare dinamicamente la modalità di ripristino e salvataggio dello stato per ciascun processo logico. Di conseguenza, da qui in poi, la modellazione dell'overhead e le ottimizzazioni autonomiche si riferiscono implicitamente a ciascun oggetto di simulazione.

3.2.1 Modello di Costo per il Sistema Non Incrementale

Per il caso non incrementale, partendo da [RA94] e ricordando le modifiche apportate, l'overhead del salvataggio e ripristino degli stati per ciascun evento può essere espresso come:

$$OH_F = \frac{S_F}{\chi_F}\delta_{LB} + P_r(S_F\delta_{RB} + \frac{\chi_F - 1}{2}\delta_e) \tag{3.1}$$

dove:

δ_e è il costo medio di esecuzione di un evento;

S_F è la dimensione media di un log completo (non incrementale);

δ_{LB} è il costo medio del salvataggio di un singolo byte appartenente all'immagine dello stato, che consideriamo includere anche il costo per byte del salvataggio dei metadati mantenuti dal gestore della mappa di memoria;

δ_{RB} è il costo medio del ripristino di un singolo byte del log, che consideriamo includere anche il costo per byte associato al ripristino dei metadati;

P_r è la probabilità di rollback (rapporto tra frequenza delle occorrenze di rollback ed esecuzioni di eventi);

χ_F è l'intervallo di log utilizzato quando si sta operando in modalità non incrementale, che determina quindi la lunghezza attesa della fase di coasting forward.

Ricordando il risultato in [RA94] e l'equazione 1.2, l'overhead appena descritto viene minimizzato quando $\chi_F = \left\lceil \sqrt{2P_r \frac{\delta_{LB} S_F}{\delta_e}} \right\rceil$, e chiamiamo pertanto χ_F^{opt} l'intervallo ottimo di salvataggio dello stato non incrementale, secondo questa equazione.

3.2.2 Modello di Costo per il Sistema Incrementale

Per quanto riguarda la modalità incrementale, così come supportata dall'architettura qui presentata, le operazioni di log non richiedono assolutamente di essere svolte prima della simulazione di ciascun evento, ma possono essere eseguite periodicamente, rifacendosi allo schema SSS presentato nel paragrafo 1.2.2 in una variante incrementale. Infatti, esse sono basate sul riconoscimento delle porzioni di memoria toccate in scrittura a partire dall'ultimo log, indipendentemente dalla quantità di eventi che hanno effettivamente modificato lo stato.

In maniera del tutto analoga, la ricostruzione dello stato a partire da qualsiasi tempo di simulazione viene supportata da un misto di ripristino dello stato dal log e di classico coasting forward. Inoltre, log completi (non incrementali) possono essere interallacciati ai log incrementali per assicurare la fossil collection[1]. Questi log completi possono essere utilizzati durante le procedure di ripristino dello stato dal momento che, mentre si scandisce la

[1] L'operazione di memorizzare un log completo dello stato, tuttavia, può essere svolta con una frequenza estremamente bassa.

catena di log all'indietro, l'operazione di ripristino dell'immagine completa dello stato può essere finalizzata estraendo dal log completo tutte quelle porzioni dello stato non ancora ripristinate.

Per tenere in considerazione tutti questi meccanismi interni forniti dal gestore della mappa di memoria, l'equazione 3.1 può essere adattata come segue:

$$\begin{aligned} OH_I &= \frac{S_P}{\chi_I}\delta_{LB} + \frac{(S_F - S_P)}{\chi_I \chi_{I,F}}\delta_{LB} + \\ &\quad P_r\left[S_F\delta_{RB} + \frac{\chi_I - 1}{2}(\delta_e + \delta_m)\right] + \delta_m \end{aligned} \tag{3.2}$$

dove i termini aggiuntivi o differenti hanno i seguenti significati:

S_P è la dimensione media di un log parziale (incrementale);

χ_I è l'intervallo di log utilizzato quando si sta operando in modalità incrementale, che determina quindi la lunghezza attesa della fase di coasting forward;

$\chi_{I,F}$ è il passo di interleaving tra log completi e log incrementali (ossia, il numero di operazioni di log incrementale dopo le quali viene memorizzato un log completo);

δ_m è il costo per eseguire la routine di tracciamento delle scritture in memoria (chiamata `monitor`) e presentata nel paragrafo 2.2.7.

Nell'equazione 3.2, il termine $S_F\delta_{RB}$ che prende in considerazione il costo di un ripristino dello stato da un log — che viene pagato in caso di rollback — è lo stesso dell'equazione 3.1. Ciò è giustificato dal meccanismo precedente, secondo il quale tutti i chunk in uso che appartengono all'immagine dello stato sono ripristinati (recuperandoli o dalla catena di log incrementali, o dal primo log completo trovato durante la procedura di scansione della catena).

Ad ogni evento, inoltre, viene addebitato il costo della routine di monitoring δ_m, che compare anche durante il coasting forward. Ricordando nuovamente il risultato in [RA94] e l'equazione 1.2 per la minimizzazione dell'overhead rispetto all'intervallo di log, otteniamo che il valore ottimo nel caso della modalità incrementale può essere calcolato come $\chi_I = \left\lceil \sqrt{2P_r \frac{\delta_{LB} S_P}{\delta_e + \delta_m}} \right\rceil$, e chiamiamo χ_I^{opt} l'intervallo ottimo secondo quest'equazione.

Inoltre, dai risultati di benchmark in [VPQ09], un giusto valore per $\chi_{I,F}$, che non aggiunge un overhead addizionale a causa dei log completi, e che assicura allo stesso tempo un ripristino efficiente della memoria durante la fossil collection, è nell'ordine dei 10. È stato utilizzato questo valore nella configurazione del sottosistema autonomico.

3.2.3 Modello Integrale di Costo Complessivo

Dall'analisi condotta nei paragrafi 3.2.1 e 3.2.2 si ottiene, per ciascuno dei modelli di salvataggio dello stato, la descrizione del costo e l'indicazione del valore ottimo dei corrispondenti parametri indipendenti, ossia degli intervalli di log. Nell'architettura autonomica di salvataggio e ripristino dello stato che presento in questo lavoro, per selezionare la modalità operativa più efficiente sarebbe sufficiente scegliere, una volta identificato il miglior valore di intervallo di log, quello con il più basso overhead atteso.

Tuttavia, viene scelta come modalità migliore quella che offre la più alta performance, tenendo anche in considerazione le possibili fluttuazioni riguardanti i parametri dei modelli che non possono essere controllati in maniera diretta, poiché dipendenti dalle dinamiche di esecuzione ottimistica del modello di simulazione[2]. Un approccio di questo tipo mira a fornire stabilità in maniera proattiva al sistema: è questa una caratteristica fortemente richiesta ad un sistema autonomico, come descritto nel paragrafo 3.3.

Il meccanismo autonomico di selezione della modalità operativa migliore, quindi, si basa sulla funzione di costo $CF(\chi_F^{opt}, \chi_I^{opt})$ definita come:

$$CF(\chi_F^{opt}, \chi_I^{opt}) = OH_F(\chi_F^{opt}) - OH(\chi_I^{opt}) \tag{3.3}$$

e sul risultato dell'integrazione di questa funzione di costo su un dominio multidimensionale definito dai valori dei parametri (δ_e, δ_m, δ_{LB}, δ_{RB}, P_r, S_F, S_P). La funzione integrale consente di prendere in considerazione le possibili fluttuazioni nei parametri che vengono misurati per valutare la funzione di costo. Per ogni parametro x che definisce una dimensione del dominio di integrazione, la funzione di costo viene integrata sull'intervallo $\bar{x} \pm \alpha\bar{x}$, in cui suggerisco $\alpha = 0.1$ per catturare fluttuazioni del parametro statisticamente rilevanti.

[2]Si tratta di tutti i parametri che compaiono nei modelli, fatta eccezione per gli intervalli di log χ_F e χ_I (o $\chi_{I,F}$) che possono essere ottimizzati analiticamente durante l'esecuzione.

Se il risultato dell'integrazione è negativo, allora la modalità operativa scelta è quella non incrementale (con l'intervallo di salvataggio dello stato impostato a χ_F^{opt}), altrimenti la modalità selezionata è quella incrementale (con l'intervallo di salvataggio dello stato impostato a χ_I^{opt}). Assumendo l'indipendenza dei parametri che definiscono il dominio di integrazione — il che è una scelta ragionevole dal momento che i valor medi $\bar{x}_i$ sono determinati in maniera operativa con un campionamento diretto del processo stocastico corrispondente, cfr. paragrafo 3.3.1 — la funzione integrale per $CF(\chi_F^{opt}, \chi_I^{opt})$ è un polinomio. Questo risultato consente una valutazione della funzione estremamente semplice ed efficiente. Esso ha la forma:

$$\left(\frac{2\alpha S_F^2}{\chi_F^{opt}} - \frac{2\alpha S_P^2}{\chi_I^{opt}} - \frac{2\alpha S_F^2 - 2\alpha S_P^2}{\chi_I^{opt}\chi_{I,F}}\right) 2\alpha\delta_{LB}^2 + 2\alpha Pr^2 \left(\frac{\chi_F^{opt} - \chi_I^{opt}}{2} 2\alpha\delta_e^2 - \frac{\chi_I^{opt} - 1}{2} 2\alpha\delta_m^2\right) - 2\alpha\delta_m^2 \tag{3.4}$$

Posto che, per determinare quale sia la modalità d'esecuzione migliore è d'interesse soltanto il segno di questa funzione di costo, si può ulteriormente semplificare l'equazione dividendo per un fattore 2α, così da ridurre l'overhead del calcolo del modello di alcune istruzioni macchina in virgola mobile, necessarie per il calcolo delle moltiplicazioni. Il modello di costo finale, pertanto, risulterà essere:

$$2\alpha \left[\left(\frac{S_F^2}{\chi_F^{opt}} - \frac{S_P^2}{\chi_I^{opt}} - \frac{S_F^2 - S_P^2}{\chi_I^{opt}\chi_{I,F}}\right)\delta_{LB}^2 + Pr^2\left(\frac{\chi_F^{opt} - \chi_I^{opt}}{2}\delta_e^2 - \frac{\chi_I^{opt} - 1}{2}\delta_m^2\right)\right] - \delta_m^2 \tag{3.5}$$

3.3 Sistema Autonomico di Ottimizzazione

Il modello e la modalità di selezione presentati finora richiedono la definizione di una *guardia* che invochi il ricalcolo e la valutazione della funzione integrale durante l'esecuzione, in modo tale da effettuare dinamicamente la riselezione della modalità di salvataggio dello stato migliore.

Nel sistema autonomico che sto presentando, assumo che l'esecuzione della simulazione sia partizionata in una fase di inizializzazione/avvio ed in una fase a regime. Durante la fase di startup, una delle due modalità viene scelta come predefinita, e viene mantenuta in esecuzione fino al termine

della fase stessa. In seguito, prima di entrare nella fase a regime, la funzione integrale viene valutata in base alle misurazioni dei parametri effettuate durante la fase iniziale.

Per migliorare l'efficienza, il valor medio dei parametri viene calcolato in maniera incrementale (ossia, senza la necessità di memorizzare ciascuno dei campioni individuali, e quindi senza consumare memoria).

Una volta che si è scelta l'iniziale modalità d'esecuzione migliore, le successive riselezioni vengono effettuate quando scatta una guardia, basata sul valor medio corrente di uno qualsiasi dei parametri $\bar{x}_i$ e sui valori $\bar{x}_i^*$ e $\alpha\bar{x}_i^*$ utilizzati durante l'ultima selezione della modalità. Se, per qualsiasi parametro x_i l'espressione $|\bar{x}_i - \bar{x}_i^*| > \alpha\bar{x}_i^*$ risulta verificata durante l'esecuzione, allora la funzione integrale viene ricalcolata sulla base dei valor medi correnti.

La ragione alla base di una soglia di questo tipo sta nel fatto che la precedente selezione dinamica della modalità di log migliore è stata svolta sulla base dei parametri statistici $\bar{x}_i^*$ e $\alpha\bar{x}_i^*$, i quali non possono più essere considerati rappresentativi delle dinamiche d'esecuzione correnti. Nel caso in cui la media corrente esca dall'intervallo di integrazione del corrispondente parametro, risulta probabile che si siano verificate alcune variazioni nelle dinamiche d'esecuzione, il che genera la necessità di rivalutare la decisione circa la modalità d'esecuzione migliore del salvataggio dello stato.

Come ultima osservazione, sottolineo che invece di utilizzare la media aritmetica, il sistema si basa sulla media mobile esponenziale $S_t = \alpha Y_{t-1} + (1-\alpha)S_{t-1}$ con un valore di $\alpha = 0.1$, che fornisce una reattività migliore rispetto alle variazioni del corrispondente processo stocastico.

3.3.1 Monitoraggio dei Parametri

Come è stato mostrato, l'approccio proposto si basa sui valor medi dei parametri che appaiono nei modelli presentati nelle equazioni 3.1 e 3.2, valori che vengono utilizzati per definire i bordi nel dominio di integrazione multidimensionale. Il sistema si basa su un processo di campionamento a runtime per calcolare la media di ciascuno dei parametri

Una difficoltà rilevante in questo processo di campionamento risiede nel fatto che il valor medio di ogni parametro x_i che appare nei modelli richiede il tracciamento del processo durante il tempo d'esecuzione, indipendentemente dalla modalità di salvataggio dello stato, incrementale o non, che è a regime.

Ciò è dovuto al fatto che la media viene utilizzata sia come guardia, sia nel processo di riselezione della modalità.

In questo senso, i parametri δ_m e S_P, utilizzati per valutare il costo della modalità di salvataggio dello stato incrementale, devono essere campionati anche quando è operativa la modalità non incrementale.

Nei paragrafi seguenti verranno mostrati schemi ad hoc utilizzati per effettuare tali misurazioni. Per quanto riguarda il parametro P_r, esso viene valutato con schemi tradizionali, come quello presentato in [RA94], basato sul conteggio del numero di rollback avvenuti durante un dato intervallo di eventi eseguiti.

Costo degli Eventi e del Tracciamento degli Accessi in Memoria

Per determinare il costo degli eventi e del tracciamento degli accessi in memoria, il sistema autonomico qui presentato implementa un meccanismo di campionamento basato sul servizio `gettimeofday()`, che offre una granularità al millisecondo. L'invadenza di questo approccio è trascurabile, dal momento che esso ha un overhead di meno di un millisecondo su macchine convenzionali, considerando entrambe le chiamate necessarie a memorizzare il tempo di inizio e di fine dell'intervallo che definisce il campione di latenza che si vuole valutare.

L'approccio utilizzato si basa su una conoscenza a priori del costo $\delta_{tracking}$ per l'esecuzione di una singola istanza della routine di tracciamento degli accessi, che dipende dal numero di istruzioni macchina necessarie a monitorare un accesso in scrittura a memoria, e dalla velocità del processore[3]. Inoltre, nell'architettura è stato inserito un modulo specifico di benchmark che determina, in maniera del tutto trasparente all'utente, la latenza della routine di tracciamento delle scritture nel momento in cui il software viene installato su una specifica piattaforma hardware.

All'interno di ciascun processo logico viene utilizzato un contatore *Count* per determinare il numero di chiamate alla routine di tracciamento degli accessi, che avvengono durante l'esecuzione di ciascun evento. Nel caso in cui la modalità corrente d'esecuzione sia incrementale, i moduli di livello

[3]Il non determinismo nella quantità di istruzioni da eseguire durante l'esecuzione della routine di monitoraggio degli accessi, dovuta eventualmente ad operazioni di ricerca iterativa all'interno della tabella di metadati, viene resa statisticamente non rilevante grazie ad una cache software che, dato l'indirizzo toccato dalla scrittura in memoria, consente un mapping diretto verso la riga della tabella di metadati relativa al chunk toccato, come mostrato in [PVQ09].

applicativo — la cui esecuzione viene attivata dall'invocazione della corretta funzione di callback (secondo lo schema di coesistenza delle versioni multiple presentato nel paragrafo 3.1) — nascondono al loro interno le chiamate alla funzione `monitor`, che incrementa *Count* ogni volta che essa viene eseguita.

Chiamando Δ_e^i il wall-clock-time tra l'inizio e la fine dell'esecuzione dell'i-esimo evento di un particolare oggetto di simulazione, valutato tramite il servizio `gettimeofday()`, otteniamo:

$$\delta_e^i = \Delta_e^i - \delta_m^i \tag{3.6}$$

dove δ_e^i denota il campione i-esimo del costo dell'evento, e δ_m^i il campione i-esimo del costo del tracciamento delle scritture a memoria, che può essere calcolato come $Count^i \cdot \delta_{tracker}$.

L'equazione 3.6 si riduce a $\delta_e^i = \Delta_e^i$ nel caso in cui il layer di salvataggio dello stato stia adottando la modalità non incrementale, dal momento che la versione corrispondente del codice applicativo non effettua alcuna chiamata alla routine di tracciamento degli accessi a memoria.

Tuttavia, come indicato precedentemente, vi è la necessità di campionare il valore δ_m^i anche nel caso in cui la modalità di salvataggio dello stato sia non incrementale. Per fornire il valore di δ_m^i anche in quest'altra situazione, la procedura di generazione delle due versioni del codice applicativo è stata leggermente modificata, in modo tale che il codice che supporta l'esecuzione in modalità non incrementale nasconda al suo interno una forma estremamente leggera di instrumentazione, in cui ciascuna istruzione di scrittura in memoria viene preceduta da una singola istruzione assembly `ADD-r/m32,imm8`, che consente l'aggiornamento del contatore *Count*. In questo modo è possibile inferire il valore di δ_m^i semplicemente moltiplicando l'i-esimo campione del valore del contatore per il costo già noto $\delta_{tracking}$, esattamente come nel caso incrementale.

È importante sottolineare come questo schema di instrumentazione inserisca un overhead trascurabile, grazie all'approccio a singola istruzione, che pertanto non altera la validità del modello di overhead dell'equazione 3.1 che esclude costi associati all'instrumentazione all'interno del codice applicativo.

Occorre anche notare che meccanismi basati su clock a real time acceduti dal servizio `gettimeofday()` si adattano a scenari in cui le piattaforme di calcolo sono dedicate interamente all'esecuzione della simulazione, uno

scenario tipico in situazioni in cui la performance è un fattore critico. In caso di time-sharing con altre applicazioni, un approccio del genere deve essere modificato tramite l'integrazione di soluzioni basate sulla pre-analisi del codice e sul profiling a runtime, come quelle discusse in [Qua01].

Dimensione dei Log Completi e Parziali

I campioni S_F^i della dimensione dei log non incrementali possono essere catturati immediatamente dal sistema autonomico indipendentemente dalla modalità di salvataggio dello stato che è attiva, dal momento che il gestore della mappa di memoria mantiene dei metadati (nello specifico, un accumulatore) che registra, in ogni istante, l'occupazione reale di memoria dell'immagine di stato degli oggetti di simulazione (in termini di quantità di byte associati ai chunk correntemente allocati). Pertanto, i campioni S_F^i possono essere recuperati semplicemente richiedendo al gestore della memoria dinamica il valore dell'accumulatore. Nell'implementazione sviluppata, il sottosistema autonomico effettua questa richiesta ogni volta che un log (incrementale o meno) viene memorizzato.

Un approccio differente è invece richiesto per catturare i campioni S_P^i dei log parziali (incrementali). In particolare, quando la modalità di salvataggio dello stato attiva è quella incrementale, il gestore della mappa di memoria aggiorna un secondo accumulatore che tiene traccia della quantità di byte associati ai chunk marcati come sporchi a partire dall'ultima operazione di log. Il valore di questo accumulatore, pertanto, viene utilizzato direttamente come valore di S_P^i quando la modalità di salvataggio dello stato incrementale è attivata.

Qualora, invece, la modalità attivata sia quella non incrementale, l'accumulatore appena citato non viene aggiornato e pertanto, il valore S_P^i viene inferito come segue. Ogni $K \cdot \chi_F^{opt}$ operazioni di logging non incrementale, l'oggetto di simulazione viene obbligato, dopo l'esecuzione dell'evento successivo, a comparare, un chunk per volta, l'immagine di memoria corrente con quella che era stata salvata nell'ultimo buffer di log. Il confronto viene svolto soltanto tra quei chunk che appartengono ad entrambe le immagini, prendendo pertanto in considerazione la porzione di stato che è stabile tra due checkpoint successivi.

Ovviamente, il costo di questa operazione dipende da K e dalle specifiche ottimizzazioni della comparazione tra ciascuna coppia di chunk. Per

quanto riguarda il secondo aspetto, si è deciso di non utilizzare il tradizionale servizio `memcmp()` dal momento che, essendo dipendente dalla sua implementazione, potrebbe non supportare una fermata anticipata non appena venga individuata la prima differenza tra i chunk. Il sistema contiene pertanto al suo interno dei moduli assembly ad hoc che effettuano iterativamente il confronto tra aree di memoria, sfruttando al massimo le dimensioni dei registri più grandi disponibili nella CPU ad ogni passo della comparazione, e che implementano una procedura di *early stop* non appena viene identificato un singolo byte di differenza tra due chunk. Questa politica si sposa bene con la granularità al chunk offerta dal gestore della mappa di memoria e dal sottosistema di salvataggio dello stato.

In aggiunta, i moduli in questione ottimizzano il tentativo di arrestare il prima possibile il confronto in caso di differenze tra chunk, seguendo il seguente schema. I chunk di taglia piccola sono confrontati partendo dal primo byte, fino ad arrivare all'ultimo. Quando ci si imbatte in chunk di taglia più grande, invece, la procedura controlla i byte in maniera non continua, partendo dall'inizio del chunk stesso e dai 3/4 della sua dimensione. Questo approccio segue una pratica comune di programmazione, secondo la quale le strutture dati sono organizzate in maniera tale che i campi aggiornati più di frequente sono posti all'inizio ed alla fine della struttura stessa (come ad esempio, puntatori per collegare nodi di liste sparsi all'interno della memoria). Pertanto, controllando l'inizio del chunk è più probabile trovare aree di memoria modificate. Allo stesso tempo, partendo da 3/4 dell'area, si tiene in considerazione anche la possibile frammentazione interna del chunk, dovuta alla tipica non correlazione tra dimensione di una struttura e area di memoria che la contiene[4].

Una volta identificati i chunk sporcati, secondo questo schema, la corrispondente percentuale p di byte sporchi viene riportata alla dimensione totale dello stato S_F^i per generare il j-simo campione S_P di stato parziale, calcolato come $S_P^j = p \cdot S_F^i$.

Per quanto riguarda il valore di K — ossia il secondo fattore che determina l'overhead finale della stima dei campioni S_P quando la modalità di salvataggio dello stato non incrementale è attiva — il sottosistema autonomico utilizza un approccio statico dove K assume il valore di 20. Dato

[4] I chunk, in DyMeLoR, sono preallocati con dimensioni progressive, secondo le potenze di 2.

che il costo associato alla procedura di stima di un singolo campione è, nel caso peggiore, confrontabile con quello di un'operazione di salvataggio di un log completo[5], questa scelta può far aumentare l'overhead reale, quando la modalità non incrementale è a regime, al più di un fattore 5%.

Secondo le considerazioni appena fatte, pertanto, si prevede che l'overhead aggiunto per determinare il valore dei campioni di S_P, quando la modalità non incrementale è operativa, sia trascurabile, non alterando pertanto la validità del modello di overhead non incrementale mostrato nell'equazione 3.1.

Costi di Salvataggio/Ripristino per byte

Gli ultimi parametri coinvolti nel processo di campionamento sono i costi per byte delle operazioni di salvataggio e ripristino dello stato, ossia δ_{LB} e δ_{RB}. Tuttavia, δ_{RB} non appare nella formula finale, pertanto ci si può concentrare su δ_{LB}.

Per campionare δ_{LB}, viene nuovamente utilizzato il servizio `gettimeofday()`, in combinazione con il procedimento di campionamento di S_F ed S_P mostrato nel paragrafo precedente. In particolare, la latenza dell'i-sima operazione di salvataggio dello stato, chiamata Δ^i_{log}, viene misurata tramite `gettimeofday()` ed in seguito normalizzata o sul corrispondente campione S_F, o sul corrispondente campione S_P, a seconda della modalità operativa correntemente attiva.

Dato che Δ^i_{log} tiene conto anche della manipolazione e del salvataggio dei metadati associati ai chunk che vengono memorizzati, la normalizzazione consente di catturare nei campioni di δ_{LB} anche il costo del processamento di tali metadati, riportandoli al costo del salvataggio di un singolo byte.

3.3.2 Confronto finale

Come mostrato in tabella 3.1, il sottosistema autonomico riesce ad offrire tutte le funzionalità già proposte in DyMeLoR e Di-DyMeLoR. In aggiunta, tramite l'utilizzo delle tecniche mostrate in questo capitolo, si propone come uno dei sottosistemi di salvataggio degli stati capace di offrire il

[5]Le operazioni di confronto della memoria, infatti, sono simili come costo a quelle di copia della memoria, dal momento che coinvolgono gli stessi spostamenti tra memoria e registri. Inoltre, la procedura di *early stop* sfavorisce un incremento della latenza nelle operazioni di confronto.

maggior numero di caratteristiche, rispetto a quelli proposti fino ad oggi in letteratura.

Come verrà mostrato nel capitolo 4, il sottosistema autonomico offre un incremento di prestazioni che non è dipendente dalla modalità d'esecuzione del modello. Inoltre, permette al programmatore del livello applicativo di poter utilizzare tutti i costrutti ANSI-C, inserendosi in modo trasparente tra il software di simulatione ed il kernel della piattaforma.

State Saving	Fossil Collection	Coasting Forward	Checkpointing Periodico	Periodo Adattativo	Stato Dinamico	Incrementalità	Granularità Arbitraria	Trasparenza	Instrumentazione	Granularità degli Eventi	Multiplexing	Modello Analitico	Misure Temporali	Euristiche	Autonomicità	Stabilità alle Fluttuazioni
[Jef85]																
[Jef90]	✓															
[Bel92]	✓	✓	✓													
[PW93]	✓	✓	✓	✓								✓	✓			
[RA94]	✓	✓	✓	✓								✓				
[SR96]	✓	✓	✓	✓						✓		✓				
[FW95]	✓	✓	✓	✓									✓	✓		
[Qua98]	✓	✓	✓	✓						✓		✓				
[BS93]	✓	✓				✓	✓									
[SQ05a]	✓	✓				✓		✓								
[RLAM96]	✓	✓				✓										
[WP96]	✓	✓	✓	✓		✓		✓	✓				✓			
[GUCF97]	✓	✓		✓		✓					✓					
[Qua01]	✓	✓	✓	✓						✓		✓				
DyMeLoR	✓	✓	✓	✓	✓			✓								
Di-DyMeLoR	✓	✓	✓	✓	✓	✓	✓	✓	✓				✓	✓		
Autonomico	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓

Tabella 3.1: Sottosistema autonomico a confronto

Capitolo 4

Risultati Sperimentali

In questo capitolo vengono riportati i risultati sperimentali che consentono una valutazione dell'architettura autonomica di salvataggio e ripristino degli stati.

Il modello di simulazione di prova rappresenta la copertura GSM lungo il Grande Raccordo Anulare, l'autostrada ad anello che circonda la città di Roma. La lunghezza del Grande Raccordo Anulare è di 68 Km e la connettività GSM viene garantita tramite 8 celle GSM, ciascuna delle quali offre una copertura fino a 9 Km lungo l'autostrada. Secondo quanto fornito dalla compagnia telefonica, ogni cella ospita 1000 canali radio [pri]. In figura 4.1 viene mostrato uno schema di come sono distribuite le celle. Nelle simulazioni svolte, i canali di comunicazione sono modellati con alta fedeltà, attraverso la simulazione esplicita della regolazione e dell'utilizzo della potenza e tenendo conto di fenomeni di interferenza e di attenuazione, sulla base dello stato corrente della cella e prendendo in considerazione anche le condizioni meteorologiche. Le celle GSM lungo l'autostrada ad anello servono principalmente le richieste di comunicazione di dispositivi mobili a bordo di veicoli che sfrecciano lungo l'autostrada.

L'interesse nei confronti della simulazione di questo tipo di sistema è legato alla necessità di valutare se il dimensionamento corrente è adeguato a supportare sia condizioni di carico di lavoro normale, sia situazioni di picchi di carico dovute a periodi di alto utilizzo dell'autostrada, che possono portare ad ingorghi di traffico. Allo stesso tempo, valutare la disponibilità di canali radio e determinare il consumo di potenza per servire aree suburbane poste vicino all'autostrada anche in caso di picchi di traffico è un altro aspetto di interesse.

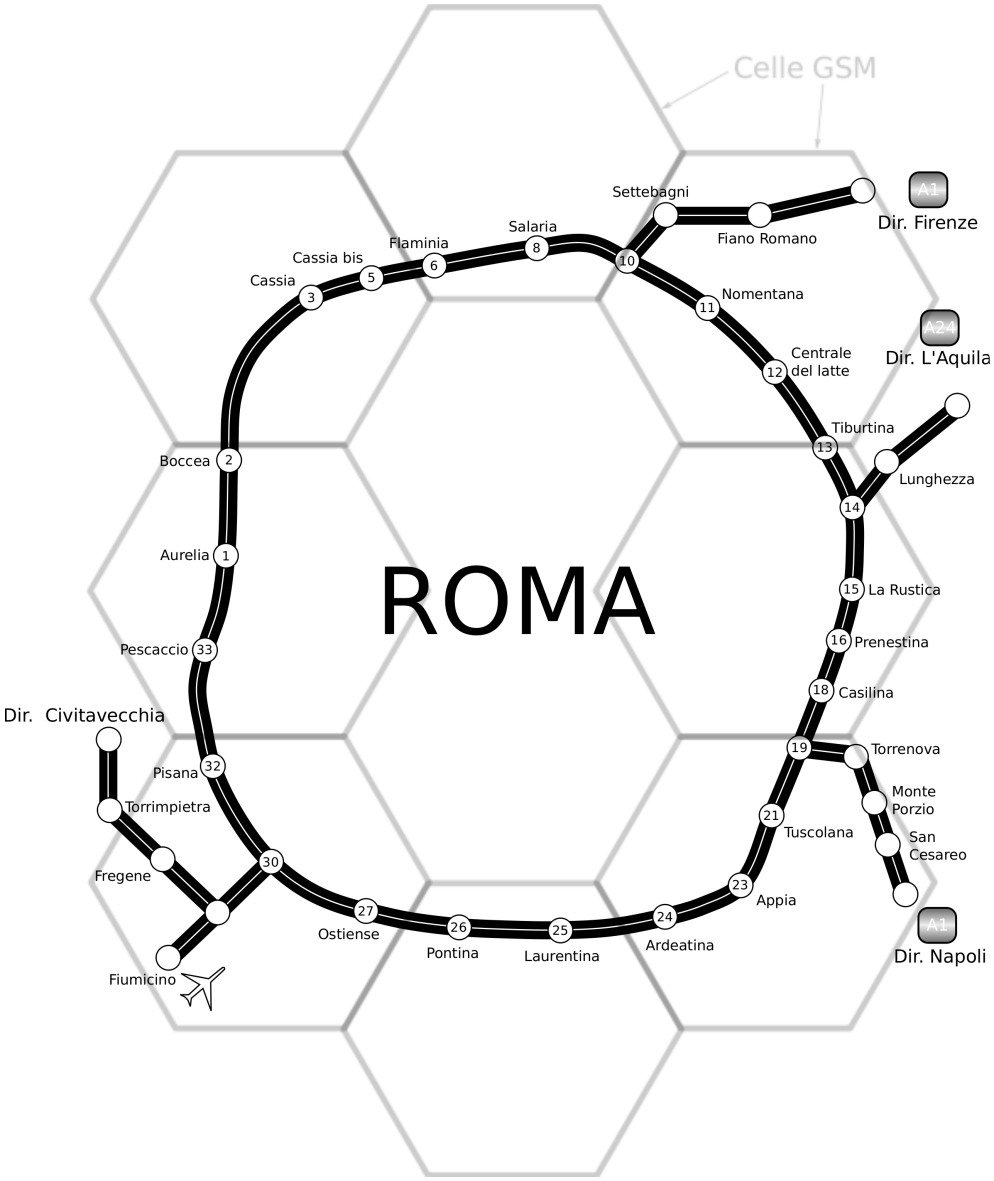

Figura 4.1: Configurazione della simulazione

Il modello di regolazione della potenza è stato implementato seguendo i risultati in [KB02]. In particolare, ciascuna cella GSM tiene traccia, tramite strutture dati allocate dinamicamente, dell'assegnazione dei canali e della gestione della potenza relativa alle chiamate in corso. Quando una chiamata destinata ad un dispositivo mobile ospitato da una data cella viene istanziata, viene creato, tramite strutture dati allocate dinamicamente, un record di gestione della chiamata che viene collegato ad una lista di record già attivi. Ciascun record viene rilasciato quando la chiamata corrispondente termina, o quando viene trasferita verso una cella adiacente. In questo secondo caso, un'analoga procedura di istanziazione di chiamata viene eseguita nella cella che riceve il dispositivo.

Quando la chiamata è istanziata, viene effettuata anche la regolazione della potenza, che consiste nella scansione della lista di record sopraccitata per il calcolo della minima potenza di trasmissione che consenta alla chiamata in arrivo di raggiungere il valore di soglia del SIR, secondo quando dettato dalla tecnologia GSM. Inoltre, vengono anche aggiornate le strutture dati che tengono traccia dei coefficienti di attenuazione, secondo un modello meteorologico che descrive le variazioni climatiche rilevanti intorno alla città di Roma. La granularità minima con la quale il modello prende in considerazione queste variazioni climatiche è di 10 secondi.

È stata simulata un'intera settimana del sistema di copertura GSM lungo l'autostrada, prendendo in considerazione esplicitamente le variazioni giornaliere del traffico. Statistiche sulle variazioni del traffico sono state estrapolate da [gra]. I periodi notturni della simulazione sono caratterizzati da fattori di utilizzo prossimi allo zero (dai dati presi da [gra], durante la notte l'autostrada è percorsa da meno di 800 veicoli), mentre le ore di traffico portano a dei livelli di utilizzo nettamente maggiori. Per i giorni lavorativi della settimana, la giornata è suddivisa in un periodo notturno (circa 10 ore, dalle 21:00 alle 08:00) caratterizzato dal fattore di utilizzo minimo, mentre la parte restante della giornata è suddivisa in periodi di traffico normale e di traffico intenso (gli ingorghi hanno una durata di 2 ore, i periodi di traffico normale una durata di 4 ore).

La simulazione della parte giornaliera porta ad un aumento della frequenza di interarrivo delle chiamate τ_A per ciascuna cella, e pertanto il fattore di utilizzo dei canali aumenta. La frequenza di interarrivo subisce delle fluttuazioni calcolate secondo una distribuzione esponenziale calcolata in funzione della densità τ_A. Nello specifico, il fattore di utilizzo dei canali nelle fasi di

traffico intenso raggiunge all'incirca il 30% della capacità totale, tenendo in considerazione chiamate della durata media di 60 secondi.

Secondo le statistiche riportate in [gra], i fine settimana hanno mediamente dei carichi differenti, dal momento che esibiscono un comportamento che oscilla tra quello normale giornaliero e quello notturno lungo tutto l'arco della giornata. Statistiche più particolareggiate sulle distribuzioni delle chiamate non possono essere utilizzate a causa di problemi di privacy. Nella tabella 4.1 vengono riassunti i parametri con i quali è stata configurata la simulazione.

Il modello descritto finora è stato sviluppato per essere simulato tramite l'ambiente ROOT-Sim, in modo tale che una singola cella GSM venga modellata da un processo logico. I callback, pertanto, generano un aggiornamento dello stato delle singole celle, mentre eventi generati da un processo logico verso un altro sono essenzialmente legati ai trasferimenti di chiamata da una cella all'altra.

La piattaforma hardware utilizzata in questo studio sperimentale è una macchina QuadCore, equipaggiata con processore Intel Core 2 Quad Q6600 (supporto all'esecuzione a 64 bit, 2.4GHz, 4MB L2 Cache per coppia di core, 32KB L1 Cache per core, 1GHz Front Side Bus) e 4GB di RAM. Per quanto riguarda l'ambiente software, il Sistema Operativo è GNU/Linux (kernel 2.6.22-31 64bit, distribuzione OpenSUSE 9.2), la versione di `gcc` utilizzata è la 4.2.1, la versione di `binutils` (`ld` e `gas`) è la 2.17.50 e la versione di MPI utilizzata è OpenMPI 1.2.4.

Su questa piattaforma sono state avviate quattro istanze del kernel di simulazione di ROOT-Sim, una per ciascun core. Ciascuna istanza gestiva due celle GSM adiacenti secondo lo schema mostrato in figura 4.1.

In figura 4.2 viene mostrata la variazione del numero di eventi eseguiti e non più annullabili (*committed*) al secondo di wall-clock-time (*event rate*) ottenuta durante la simulazione di specifici intervalli di tempo virtuale, rappresentati dalla variazione del GVT sull'asse x. Questo parametro indica la velocità secondo cui un dato periodo di tempo virtuale viene simulato. Più alta è la frequenza degli eventi, più veloce è l'esecuzione durante un determinato periodo di tempo virtuale. Nel grafico sono riportate tre curve che si riferiscono (i) al caso in cui il sottosistema autonomico è attivo, (ii) al caso in cui il sottosistema viene forzato a catturare unicamente log in modalità

	Giorno			**Notte**			**Picco**		
	Macchine	τ_A	*Carico*	*Macchine*	τ_A	*Carico*	*Macchine*	τ_A	*Carico*
Leggero	100	0.5	5%	100	1	5%	400	0.25	20%
Medio	400	0.25	20%	100	1	5%	700	0.143	35%
Intenso	700	0.143	35%	100	1	5%	1000	0.1	50%

1000 canali, chiamate da 60 secondi

Tabella 4.1: Distribuzione del carico per ogni cella

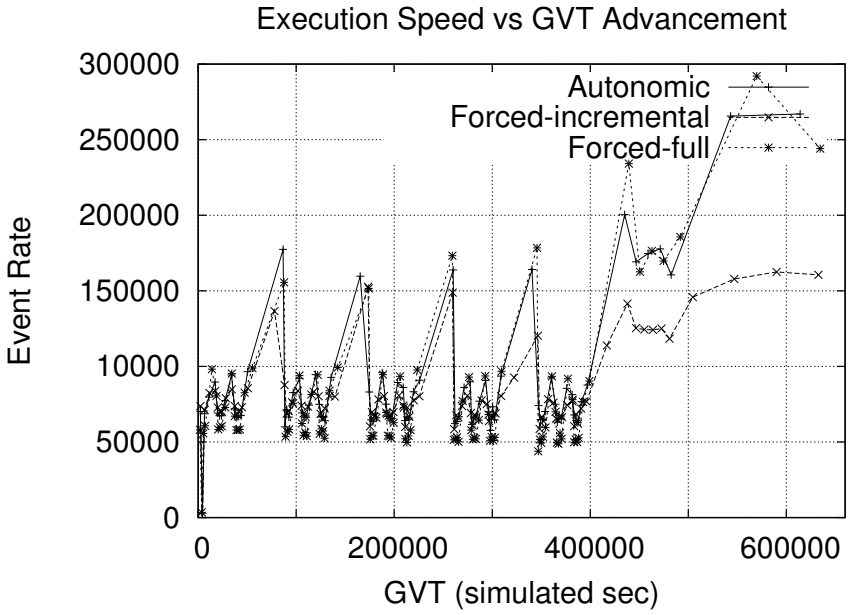

Figura 4.2: Variazione per secondo di eventi committed

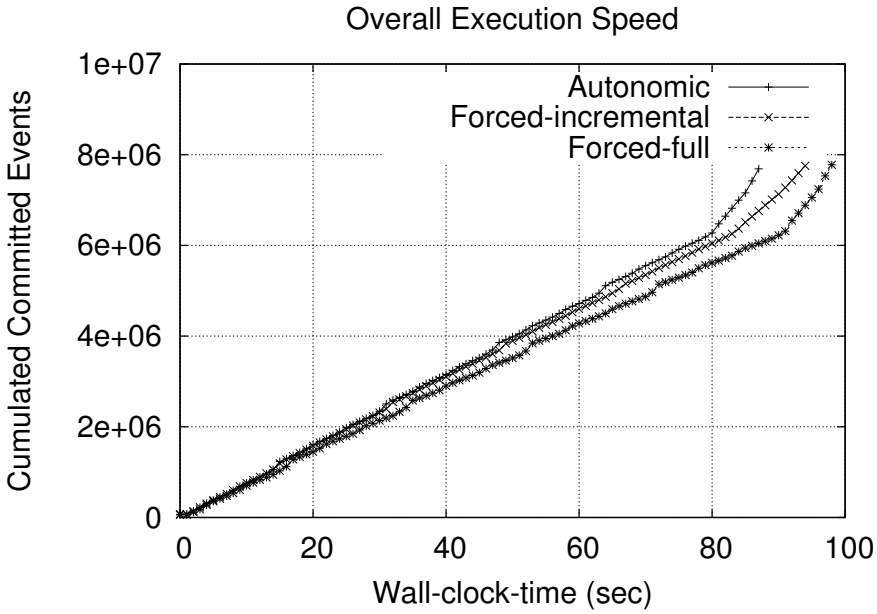

Figura 4.3: Throughput

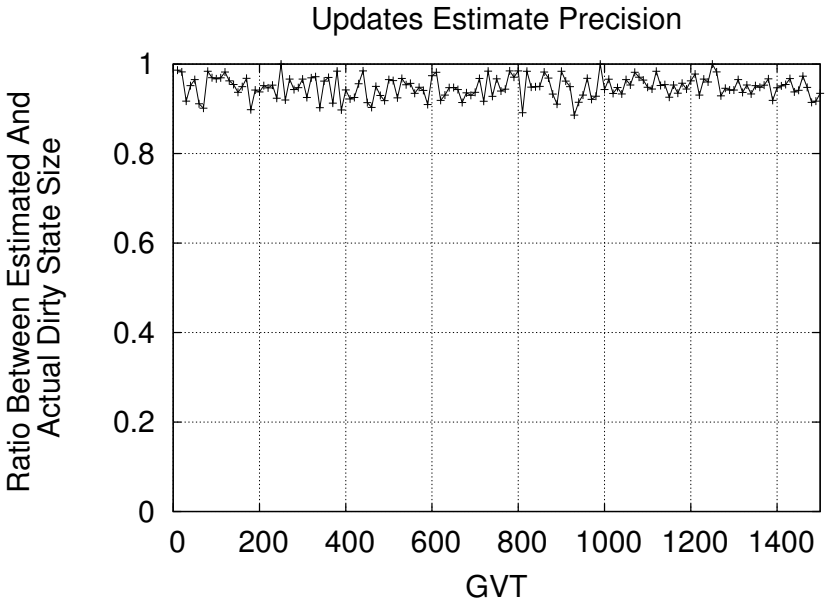

Figura 4.4: Stima della dimensione dello stato sporco

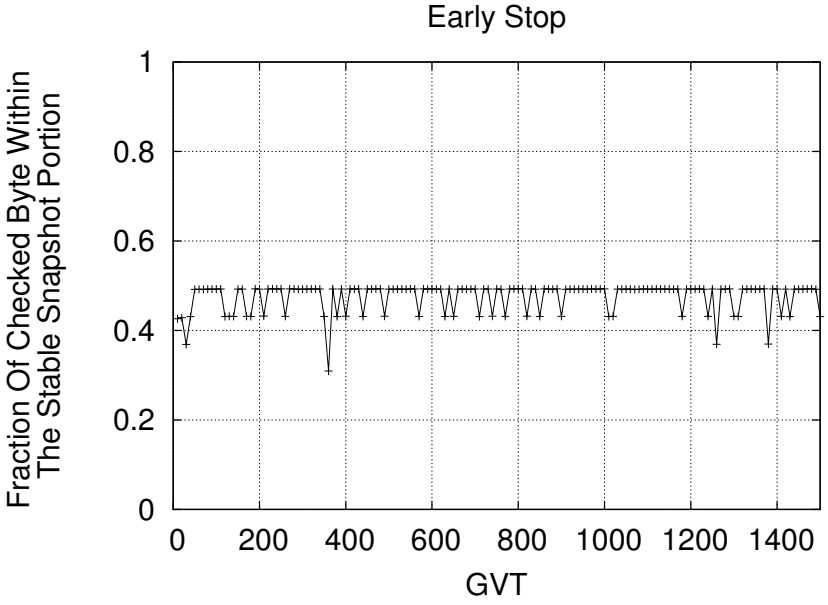

Figura 4.5: Performance della politica *early stop*

incrementale, con il valore di χ_I ottimizzato e (iii) il caso in cui il sottosistema è forzato a catturare unicamente log in modalità non incrementale con il valore di χ_F ottimizzato. Le curve per i casi (ii) e (iii) mostrano dei livelli di performance che possono essere raggiunti con l'uso di modalità di salvataggio e ripristino dello stato ottimizzato (adattivo riguardo alla selezione dell'intervallo di log) basato sulle modalità incrementale o non, ma che non permettono il passaggio da una modalità all'altra durante l'esecuzione.

Dai risultati si nota che, a seconda del periodo di simulazione (periodo notturno o periodo giornaliero) la performance migliore si ottiene utilizzando alternativamente la modalità incrementale o la modalità non incrementale. In particolare, la modalità non incrementale è più veloce quando vengono simulati i periodi notturni, mentre nei periodi giornalieri è la modalità incrementale ad essere più efficiente. Questo è un riflesso del fatto che, durante la notte e nei fine settimana, ogni cella GSM, e quindi ogni processo logico, ha uno stato di dimensione ridotta, a causa del numero minimale di record allocati per le chiamate in transito. Non è questo il caso, invece, durante i periodi giornalieri: la dimensione dello stato dei processi logici può crescere significativamente (specialmente nelle ore di traffico intenso), fino al limite di 70 KB, ed il pattern di aggiornamento dello stato quando vengono eseguiti degli eventi consente alla modalità incrementale di scavalcare come performance la modalità non incrementale, quando entrambi i periodi di salvataggio sono stati ottimizzati.

Tuttavia, il risultato più importante che si rileva dai grafici sugli event rate è quello che mostra come la configurazione autonomica cambia di modalità (incrementale e non) scegliendo sempre quella che offre una performance migliore, a seconda del periodo di simulazione corrente (giorno piuttosto che notte), tenendo pertanto in considerazione le dinamiche di esecuzione correnti (in termini di dimensione dello stato, di granularità degli eventi, di pattern di aggiornamento della memoria, ...).

L'effetto finale sulla performance di questo comportamento ottimizzato è espresso in figura 4.3, dove viene mostrato il numero accumulato di eventi completati e non più annullabili (*committed*) per unità di tempo di wall-clock per l'esecuzione della simulazione. Queste curve esprimono la capacità di ciascuna configurazione di salvataggio e ripristino degli stati di processare eventi fino al punto in cui essi non siano più annullabili (e quindi di portare avanti lavoro di simulazione utile) mentre il tempo avanza. Pertanto abbiamo una rappresentazione di quanto velocemente, rispetto al tempo di

wall-clock, il modello di simulazione viene eseguito.

Dai risultati, la capacità della configurazione autonomica di passare continuamente alla modalità migliore si riflette nel fatto che la curva del numero accumulato di eventi completati segue sempre la pendenza migliore. In altre parole, il sottosistema autonomico consente l'esecuzione del modello in modo estremamente più veloce, rispetto a ciò che accade negli altri due scenari.

In particolare, il tempo di wall-clock richiesto alla modalità autonomica per eseguire il numero di eventi necessari al completamento della simulazione, è ridotto di circa il 13% rispetto alla modalità non incrementale, e di circa il 9% rispetto alla modalità incrementale. Posto che entrambe le modalità sono eseguite con delle configurazioni fortemente ottimizzate (grazie alla riselezione dinamica degli intervalli di salvataggio dello stato) questo è un risultato estremamente importante.

Per completare questi risultati — che mostrano una visuale comprensiva delle performance che possono essere ottenute utilizzando il sottosistema autonomico presentato — vengono mostrati due grafici addizionali relativi alla bontà delle implementazioni di operazioni interne al sistema. In particolare, in figura 4.4 vengono riportati dei dati relativi alla stima della porzione sporca dello stato di un processo logico, quando l'esecuzione avanza in modalità non incrementale. Ricordo, come mostrato nel paragrafo 3.3.1, che la stima si basa sulla comparazione di chunk tra immagini dello stato successivo, prendendo in considerazione soltanto la porzione stabile dello stato. In particolare, viene mostrato il rapporto tra la dimensione stimata della porzione sporca dello stato del processo logico e la sua dimensione reale. La curva mostrata si riferisce ad un sottointervallo del tempo complessivo di simulazione, ma i dati sono comunque rappresentativi del modello nel suo insieme. Dalle curve si nota che l'errore nel processo di stima è praticamente sempre minore del 5%, con pochissimi picchi dell'ordine di non più del 10%.

In ultimo, in figura 4.5 vengono mostrati, per lo stesso intervallo di simulazione, gli effetti dell'approccio *early stop* sul confronto fra chunk (cfr. ancora il paragrafo 3.3.1). In particolare, viene riportato il rapporto tra il numero reale di byte confrontati (tra due immagini di stato consecutive) e il numero di byte totale che costituisce la porzione sporca dell'immagine in questione. Posto che l'implementazione del sistema GSM proposta colloca i campi acceduti più di frequente in cima al record, il sistema di fermata fornisce dei vantaggi reali, permettendo di effettuare un confronto solamente

su circa il 50% della porzione stabile dello snapshot.

Si vuole sottolineare che un'ottimizzazione di questo tipo può fornire dei vantaggi in termini di efficienza anche per modelli di simulazione generici che abbiano stati di dimensioni molto più grandi, posto che l'approccio citato di collocare i campi acceduti più di frequente in cima al record (abitudine, questa, di utilizzo comune nella programmazione avanzata) venga messo in atto.

Capitolo 5

Lavori Collegati

In questo capitolo discuterò alcuni lavori presentati più o meno recentemente sugli stessi ambiti di questo progetto, evidenziandone chiaramente similitudini e differenze.

Nel contesto delle simulazioni ottimistiche sono state introdotte alcune soluzioni per salvare l'intero stato di un oggetto di simulazione (al momento dell'esecuzione di un evento o dopo un certo intervallo di eventi eseguiti) [FW95, PLM94, Qua01, QS03, RA94], o per salvare incrementalmente le porzioni di stato modificate [Bru95, RLAM96, Ste93, WP96], o per supportare un misto dei due approcci [FGUC97, NF07, SE98].

Queste soluzioni hanno la necessità di (i) fornire il codice necessario per catturare gli snapshot dello stato degli oggetti all'interno del software di livello applicativo o di (ii) impiegare chiamate a funzioni tra le API di apposite librerie di checkpointing oppure di (iii) identificare staticamente (ad esempio, a tempo di compilazione) quali porzioni dello spazio di indirizzamento debbano essere considerate parte dello stato.

Di conseguenza, non viene supportata una trasparenza perfetta, dal momento che il programmatore deve necessariamente scontrarsi con le questioni legate agli snapshot degli stati. Inoltre l'identificazione statica delle locazioni di memoria da includere negli snapshot non è compatibile con l'allocazione e la deallocazione dinamica della memoria (ad esempio tramite librerie standard) a livello degli oggetti di simulazione.

È questo il caso anche del lavoro in [WP96] che ha alcune somiglianze con il mio sul piano dell'instrumentazione automatica, ma che non permette

l'utilizzo di memoria allocata dinamicamente.

La questione degli stati basati su memoria dinamica per gli oggetti di simulazione ottimistica è stata anche affrontata dai framework in [spe05, DFP+94]. Ad ogni modo, in essi vengono adoperate delle API apposite per notificare esplicitamente al kernel di simulazione che delle operazioni specifiche di allocazione o deallocazione e, più in generale, operazioni su strutture dati basate su memoria dinamica, dovranno essere ripristinabili. Pertanto, differentemente dall'approccio da me adottato, non sono supportati layout di memoria basati su servizi di allocazione e deallocazione propri dell'ANSI-C.

In termini di capacità del sottosistema di gestione della memoria, i lavori più vicini al mio approccio sono probabilmente quelli in [SQ05a, SQ05b], che presentano dei livelli software per effettuare in maniera trasparente operazioni di log e restore nelle simulazioni ottimistiche basate sullo standard di interoperabilità High-Level-Architecture (HLA).

Questi livelli si appoggiano a meccanismi di protezione della memoria propri dei Sistemi Operativi (e in particolare ai servizi offerti dalla *system call* UNIX `mprotect()`) per identificare gli aggiornamenti in memoria ed effettuare il log incrementale delle pagine sporcate che siano appartenenti ad un layout di memoria degli LP. Il kernel della piattaforma protegge da scrittura tutte le pagine che contengono gli stati degli oggetti di simulazione, così da poter intercettare i segnali di errore alzati dal kernel del Sistema Operativo ogni volta che l'applicazione effettua un'operazione di scrittura. In questo modo, ad ogni intercettazione, il kernel della piattaforma si occuperà di eseguire una copia dell'intera pagina, abilitarne la scrittura per consentire l'aggiornamento dello stato, ed infine disabilitarne nuovamente la scrittura.

Se confrontato con il mio approccio, l'overhead per tracciare gli aggiornamenti ed effettuare le operazioni di log incrementale è verosimilmente maggiore[1] ed è conveniente soltanto quando sia comparabile con il costo dei servizi di interoperabilità supportati dal middleware di HLA.

Tutto ciò rende questi approcci inadatti alle tradizionali piattaforme di simulazione ottimistica, che hanno requisiti di efficienza estremamente stringenti, e che sono l'obiettivo di questo lavoro.

[1] I continui passaggi da *user-mode* a *kernel-mode* e viceversa, necessari per cambiare i privilegi associati alle pagine e causati dai tentativi di modifiche a pagine protette da scrittura, diventano estremamente frequenti durante la simulazione, e sono operazioni intrinsecamente molto costose.

Allo stesso tempo, questo lavoro è sviluppato tenendo in considerazione motori di simulazione ottimistica più tradizionali, che si basano su un set ristretto di servizi (come ad esempio [Per05]), dove il costo delle operazioni di salvataggio e ripristino dello stato possono rappresentare un fattore dominante nei confronti della performance.

Alcuni risultati recenti [BP07, CPF99, NF07] hanno mostrato la fattibilità e l'efficienza della gestione ottimistica degli stati tramite una computazione inversa: una versione inversa del codice di simulazione di livello applicativo viene generata (automaticamente o semiautomaticamente, tramite un'analisi statica) ed utilizzata per il calcolo all'indietro, mirato al ripristino dello stato dell'oggetto di simulazione.

Questa tecnica, comunque, è più adatta per applicazioni a granularità di eventi fine, dal momento che il codice inverso è generalmente più efficiente quando a ciascun evento è associata una computazione breve. Il mio approccio, invece, tende a concentrarsi su eventi a granularità arbitraria.

Inoltre in contesti generali di simulazione (ad esempio laddove si possono verificare dei percorsi d'esecuzione non invertibili, come l'assegnazione distruttiva di variabili o la generazione di numeri pseudocasuali), questo approccio deve essere necessariamente affiancato da tecniche di log e restore simili a quelle che ho presentato in questo lavoro.

Confrontata con questi altri approcci, la mia soluzione supporta la gestione degli stati, basata su capacità di log incrementale, senza la necessità di moduli specifici per il log e per il restore all'interno del codice applicativo, né di un'interfaccia esplicita con delle librerie di log e restore. Inoltre permette di distribuire gli stati degli oggetti di simulazione su dei frammenti di memoria allocata dinamicamente, senza requisiti stringenti sulla loro dimensione. Fornisce, inoltre, un sistema ottimizzato per la selezione autonomica del migliore intervallo di salvataggio dello stato, partendo da misurazioni delle dinamiche che si verificano a tempo d'esecuzione. Infine, supporta in modo ottimizzato (e contemporaneamente) sia la modalità incrementale sia la modalità non incrementale di salvataggio dello stato, e permette l'esecuzione con qualsiasi tipo di scheduler degli eventi, indipendentemente dalla granularità di questi ultimi.

Capitolo 6

Conclusioni

In questo lavoro ho presentato il progetto e l'implementazione di un sottosistema autonomico di salvataggio e ripristino degli stati degli oggetti per sistemi di simulazione ottimistica, che rispetta tutte le specifiche richieste ad un sistema autonomico.

Il sistema in questione è un sottosistema di gestione degli stati dotato di tutte le funzioni necessarie, che consente (in maniera del tutto trasparente) di utilizzare servizi standard di allocazione/deallocazione dinamica della memoria al livello applicativo, e che supporta (sempre in maniera del tutto trasparente) modalità di salvataggio dello stato sia incrementale che non incrementale (secondo uno schema alternato) in funzione delle dinamiche d'esecuzione correnti.

Il sottosistema autonomico si basa su un tool di instrumentazione che consente la generazione di codice eseguibile in doppia versione all'interno della stessa immagine di programma, ciascuna delle quali annida o meno al proprio interno delle caratteristiche di tracciamento degli accessi in scrittura a memoria. Pertanto, ciascuna modalità di salvataggio dello stato viene supportata a runtime tramite l'esecuzione della versione di codice appropriata.

La selezione autonomica della modalità di salvataggio dello stato migliore è basata su un approccio innovativo di modellazione ed ottimizzazione, che si basa sulla capacità di catturare fluttuazioni nelle dinamiche d'esecuzione.

L'efficienza dell'approccio è stata anche testata con un caso di studio reale, collegato alla connettività della rete cellulare lungo un'autostrada.

Bibliografia

[Aya89] Rassul Ayani. A parallel simulation scheme based on the distance between objects. In *Proceedings of the SCS Multiconference on Distributed Simulation*, volume 21, pagine 113–118, 1989.

[Bel90] Steven Bellenot. Global virtual time algorithms. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pagine 122–127, Gennaio 1990.

[Bel92] Steven Bellenot. State skipping performance with the Time Warp operating system. In *Proceedings of the 6 th Workshop on Parallel and Distributed Simulation (PADS '92*, pagine 53–64, 1992.

[BP07] David W. Bauer e Ernest H. Page. An approach for incorporating rollback through perfectly reversible computation in a stream simulator. In *21st International Workshop on Principles of Advanced and Distributed Simulation*, pagine 171–178. IEEE Computer Society, 2007.

[Bru95] David Bruce. The treatment of state in optimistic systems. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pagine 40–49. IEEE Computer Society, Giugno 1995.

[BS93] Herbert Bauer e Christian Sporrer. Reducing rollback overhead in time warp based distributed simulation with optimized incremental state saving. In *Proceedings of the 26th Annual Simulation Symposium*, pagine 12–20, 1993.

[CM79] K. Mani Chandy e Jayadev Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, SE-5(5):440–452, Settembre 1979.

[CPF99] Christopher D. Carothers, Kalyan S. Perumalla, e Richard Fujimoto. Efficient optimistic parallel simulations using reverse computation. *ACM Transactions on Modeling and Computer Simulation*, 9(3):224–253, Luglio 1999.

[D'A07] Stefano D'Alessio. Tecniche di logging asincrono per piattaforme di simulazione ottimistiche. Tesi di Laurea, Computer Science Departement, DIS, Sapienza Università di Roma, Rome – Italy, 2007.

[DFP+94] Samir Das, Richard Fujimoto, Kiran Panesar, Don Allison, e Maria Hybinette. GTW: a time warp system for shared memory multiprocessors. In *WSC '94: Proceedings of the 26th conference on Winter simulation*, pagine 1332–1339. Society for Computer Simulation International, 1994.

[EAWJ96] Mootaz Elnozahy, Lorenzo Alvisi, Yi-Min Wang, e David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. Relazione tecnica, ACM Computing Surveys, 1996.

[FGUC97] Steve Franks, Fabian Gomes, Brian Unger, e John Cleary. State saving for interactive optimistic simulation. In *Proceedings of the 11th workshop on Parallel and Distributed Simulation*, pagine 72–79. IEEE Computer Society, 1997.

[FL95] Alois Ferscha e Johannes Lüthi. Estimating rollback overhead for optimism control in time warp. In *SS '95: Proceedings of the 28th Annual Simulation Symposium*, pagina 2, Washington, DC, USA, 1995. IEEE Computer Society.

[FN92] Richard M. Fujimoto e David M. Nicol. State of the art in parallel simulation. In *WSC '92: Proceedings of the 24th conference on Winter simulation*, pagine 246–254. ACM Press, 1992.

[Fuj89] Richard M. Fujimoto. The virtual time machine. In *Proceedings of the fisrt annual ACM symposium on Parallel algorithms and architectures*, pagine 199–208. ACM Press, 1989.

[Fuj90] Richard M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, Ottobre 1990.

[Fuj93] Richard M. Fujimoto. Parallel and distributed discrete event simulation: algorithms and applications. In *WSC '93: Proceedings of the 25th conference on Winter simulation*, pagine 106–114. ACM Press, 1993.

[FW95] Josef Fleischmann e Philip A. Wilsey. Comparative analysis of periodic state saving techniques in Time Warp simulators. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pagine 50–58. IEEE Computer Society, Giugno 1995.

[gra] http://traffico.octotelematics.it/.

[GUCF97] Fabian Gomes, Brian Unger, John Cleary, e Steve Franks. Multiplexed state saving for bounded rollback. In *WSC '97: Proceedings of the 29th conference on Winter simulation*, pagine 460–467, Washington, DC, USA, 1997. IEEE Computer Society.

[Inta] Intel Corporation. *IA-32 Intel(R) Architecture Software Developer's Manual Volume 1: Basic Architecture*.

[Intb] Intel Corporation. *IA-32 Intel(R) Architecture Software Developer's Manual, Volume 2A: Instruction Set Reference, A-M*.

[Intc] Intel Corporation. *IA-32 Intel(R) Architecture Software Developer's Manual, Volume 2B: Instruction Set Reference, N-Z*.

[Jef85] David R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and System*, 7(3):404–425, Luglio 1985.

[Jef90] David R. Jefferson. Virtual Time II: storage management in conservative and optimistic systems. In *Proceedings of the 9th annual ACM symposium on Principles of distributed computing*, pagine 75–89. ACM, 1990.

[KB02] Sunil Kandukuri e Stephen Boyd. Optimal power control in interference-limited fading wireless channels with outage-probability specifications. *IEEE Transactions on Wireless Communications*, 1(1):46–55, 2002.

[LL89] Yi-Bing Lin e Ed D. Lazowska. Determining the global virtual time in a distributed simulation. Relazione tecnica, University of Washington Department of Computer Science and Engineering, Gennaio 1989.

[LL90] Yi-Bing Lin e Ed D. Lazowska. *Reducing the saving overhead for Time Warp parallel simulation*. University of Washington Department of Computer Science and Engineering, Febbraio 1990.

[Mat93] Friedemann Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel Distributed Computing*, 18(4):423–434, 1993.

[NF07] Andriy Naborskyy e Richard M. Fujimoto. Using reversible computation techniques in a parallel optimistic simulation of a multiprocessor computing system. In *21st International Workshop on Principles of Advanced and Distributed Simulation*, pagine 179–188. IEEE Computer Society, 2007.

[Nic88] David M. Nicol. Parallel discrete-event simulation of FCFS stochastic queueing networks. In *PPEALS '88: Proceedings of the ACM/SIGPLAN conference on Parallel programming: experience with applications, languages and systems*, pagine 124–137, New York, NY, USA, 1988. ACM.

[Nic93] David M. Nicol. The cost of conservative synchronization in parallel discrete event simulations. *J. ACM*, 40(2):304–333, 1993.

[Nic96] David M. Nicol. Principles of conservative parallel simulation. In *WSC '96: Proceedings of the 28th conference on Winter simulation*, pagine 128–135, Washington, DC, USA, 1996. IEEE Computer Society.

[Pel07] Alessandro Pellegrini. Tracciamento trasparente ed efficiente di scritture su memoria dinamica con granularità arbitraria in architetture per il calcolo ottimistico. Tesi di Laurea, Computer Science Departement, DIS, Sapienza Univerità di Roma, Rome – Italy, 2007.

[Per05] Kalyan S. Perumalla. μ-sik: A micro-kernel for parallel/distributed simulation systems. In *PADS '05: Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*, pagine 59–68, Washington, DC, USA, 2005. IEEE Computer Society.

[PLM94] Bruno R. Preiss, Wayne M. Loucks, e Ian D. MacIntyre. Effects of the checkpoint interval on time and space in Time Warp. *ACM Transactions on Modeling and Computer Simulation*, 4(3):223–253, Luglio 1994.

[pri] Private communication.

[PVQ09] Alessandro Pellegrini, Roberto Vitali, e Francesco Quaglia. Didymelor: Logging only dirty chunks for efficient management of dynamic memory based optimistic simulation objects. In *PADS '09: Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, pagine 45–53, Washington, DC, USA, 2009. IEEE Computer Society.

[PW93] Avinash C. Palaniswamy e Philip A. Wilsey. An analytical comparison of periodic checkpointing and incremental state saving. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, pagine 127–134. IEEE Computer Society, 1993.

[QS01] Francesco Quaglia e Andrea Santoro. Semi-asynchronous checkpointing for optimistic simulation on a myrinet based now.

In *PADS '01: Proceedings of the fifteenth workshop on Parallel and distributed simulation*, pagine 56–63, Washington, DC, USA, 2001. IEEE Computer Society.

[QS03] Francesco Quaglia e Andrea Santoro. Non-blocking checkpointing for optimistic parallel simulation: Description and an implementation. *IEEE Transactions on Parallel and Distributed Systems*, 14(6):593–610, Giugno 2003.

[Qua98] Francesco Quaglia. Event history based sparse state saving in time warp. *SIGSIM Simul. Dig.*, 28(1):72–79, 1998.

[Qua99] Francesco Quaglia. Combining periodic and probabilistic checkpointing in optimistic simulation. In *PADS '99: Proceedings of the thirteenth workshop on Parallel and distributed simulation*, pagine 109–116, Washington, DC, USA, 1999. IEEE Computer Society.

[Qua01] Francesco Quaglia. A cost model for selecting checkpoint positions in Time Warp parallel simulation. *IEEE Transactions on Parallel and Distributed Systems*, 12(4):346–362, Febbraio 2001.

[RA94] Robert Rönngren e Rassul Ayani. Adaptive checkpointing in Time Warp. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, pagine 110–117. Society for Computer Simulation, Luglio 1994.

[RAT93] Hassan Rajaei, Rassul Ayani, e Lars-Erik Thorelli. The local time warp approach to parallel simulation. *SIGSIM Simul. Dig.*, 23(1):119–126, 1993.

[Rey88] Paul F. Reynolds, Jr. A spectrum of options for parallel simulation. In *Proc. of 1988 Winter Simulation Conference*, pagine 325–332. Society for Computer Simulation, Dicembre 1988.

[RLAM96] Robert Rönngren, Michael Liljenstam, Rassul Ayani, e Johan Montagnat. Transparent incremental state saving in Time Warp parallel discrete event simulation. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, pagine 70–77. IEEE Computer Society, Maggio 1996.

[SE98] Hussam M. Soliman e Adel S. Elmaghraby. An analytical model for hybrid checkpointing in Time Warp distributed simulation. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):947–951, Ottobre 1998.

[spe05] SPEEDES. *http://www.speedes.com*, 2005.

[SQ05a] Andrea Santoro e Francesco Quaglia. Transparent state management for optimistic synchronization in the High Level Architecture. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*, pagine 171–180. IEEE Computer Society, Giugno 2005.

[SQ05b] Andrea Santoro e Francesco Quaglia. A version of MASM portable across different UNIX systems and different hardware architectures. In *Proceedings of the 9th International Symposium on Distributed Simulation and Real Time Applications*. IEEE Computer Society, Ottobre 2005.

[SR96] Sven Sköld e Robert Rönngren. Event sensitive state saving in time warp parallel discrete event simulations. In *WSC '96: Proceedings of the 28th conference on Winter simulation*, pagine 653–660, Washington, DC, USA, 1996. IEEE Computer Society.

[Ste93] Jeff Steinman. Incremental state saving in SPEEDES using C Plus Plus. In *Proceedings of the Winter Simulation Conference*, pagine 687–696. Society for Computer Simulation, Dicembre 1993.

[Tel91] Gerard Tel. *Topics in Distributed Algorithms*. Cambridge University Press, 1991.

[TQ08] Roberto Toccaceli e Francesco Quaglia. Dymelor: Dynamic memory logger and restorer library for optimistic simulation objects with generic memory layout. In *PADS '08: Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation*, pagine 163–172, Washington, DC, USA, 2008. IEEE Computer Society.

[VPQ09] Roberto Vitali, Alessandro Pellegrini, e Francesco Quaglia. Benchmarking memory management capabilities within root-sim. In *DS-RT '09: Proceedings of the 2009 13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*, pagine 33–40, Washington, DC, USA, 2009. IEEE Computer Society.

[VPQ10] Roberto Vitali, Alessandro Pellegrini, e Francesco Quaglia. Autonomic log/restore for advanced optimistic simulation systems. *International Symposium on Modeling, Analysis, and Simulation of Computer Systems*, 0:319–327, 2010.

[WP96] Darrin West e Kiran Panesar. Automatic incremental state saving. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, pagine 78–85. IEEE Computer Society, Maggio 1996.

[WT09] Jun Wang e Carl Tropper. Selecting GVT interval for time-warp-based distributed simulation using reinforcement learning technique. In *SpringSim '09: Proceedings of the 2009 Spring Simulation Multiconference*, pagine 1–7, San Diego, CA, USA, 2009. Society for Computer Simulation International.

Indice analitico

S

T

V

X

Ringraziamenti

Il mio debito di gratitudine più grande va a Francesco Quaglia, per il suo aiuto lungo tutto il mio percorso della laurea magistrale e durante la stesura di questa tesi. Grazie a lui, ho scoperto l'ambito del calcolo parallelo e distribuito, cominciando a confrontarmi con le problematiche del salvataggio e ripristino degli stati di simulazione. L'opportunità di lavorare su questo ed altri progetti con lui mi ha fatto crescere moltissimo.

Ho anche un forte debito verso Roberto Vitali, eccellente compagno di lavoro, con cui sono stati ottenuti la maggior parte dei risultati in questa tesi. La sua disponibilità ha reso incredibilmente più piacevole questo ed altri lavori.

Grazie alla mia famiglia, a mio padre che mi ha sempre spronato a dare il meglio di me, a mia madre per la sua pazienza e le sue attenzioni nei miei confronti, ad entrambi per il loro amore senza il quale non potrei essere la persona che sono. Grazie a mio fratello Andrea, per aver sempre creduto in quello che faccio e per aver sempre seguito i miei passi, mentre crescevo.

Grazie a Marta, perché ha sempre sostenuto le mie scelte, illuminando ogni giorno ogni mio singolo gesto, rendendo magico ogni momento.

Grazie a tutti i miei amici che mi hanno s{o,u}pportato: a Giulia e Gian Marco, che mi sono sempre accanto in qualsiasi cosa io faccia, ai miei colleghi dell'università, che hanno sempre sostenuto ed incoraggiato tutto il lavoro, ad Enrico, per avermi aggiornato su ciò che accadeva nel mondo reale mentre ero concentrato su questo ed altri lavori e per avermi fatto sempre sognare progetti fantastici, e tutti quelli che — inevitabilmente — ho dimenticato.

Grazie ad ingegneri fantastici come Donald Knuth e Leslie Lamport, per aver lavorato duramente alla progettazione ed allo sviluppo di TEXe LATEX, senza cui questa tesi non sarebbe mai stata realizzata.

Questa tesi è realizzata utilizzando esclusivamente software open source.