Report Gruppo 27 - Progetto 2.2: Sistema di Fault-Injection per Applicazione Ridondata

Anna Lisa Maddaloni s333037, Silvia Polizzi s323914

1. Introduzione

Il sistema di Fault-Injection sviluppato consente di iniettare guasti di tipo bit-flip all'interno di un array di variabili duplicate, analizzare l'impatto dei guasti ottenuti sul comportamento dell'algoritmo di ordinamento Bubble Sort e ottenere metriche riguardanti l'overhead di memoria e di CPU time.
In particolare, il sistema viene configurato elaborando da un file `.toml` in input gli opportuni parametri: il numero di elementi dell'array su cui eseguire l'ordinamento, il numero di fault da iniettare, la posizione massima del bit più significativo su cui si può eseguire il flip e il tempo massimo di iniezione.
I guasti vengono generati in maniera randomica e registrati mediante la struttura dati "Fault": a ciascun guasto corrisponde l'indice dell'array da manipolare, il bit da manipolare e il tempo di attesa.
Per ogni guasto, viene creato un array randomico ridondato mediante la duplicazione delle variabili, che fa riferimento alla struttura dati "`Redundant<T>`".
Viene utilizzato un approccio multi thread, dove, per ogni fault da testare, un thread separato si occupa di iniettare l'errore nel dato, mentre un altro thread esegue il sorting sull'array. Per garantire un accesso sicuro ai dati condivisi, l'array è protetto da un mutex, che impedisce modifiche concorrenti. Inoltre, per sincronizzare i due thread e assicurare che l'iniezione del fault avvenga durante l'esecuzione del sorting, viene utilizzata una **barriera di sincronizzazione**. Questo permette di coordinare l'inizio delle operazioni, evitando che il sorting termini prima che il fault venga iniettato, assicurando così test affidabili sulle effettive conseguenze degli errori introdotti.
I risultati dei test vengono analizzati e registrati tramite un oggetto "Analyzer", che raccoglie informazioni sul successo o il fallimento dell'ordinamento, nonché eventuali malfunzionamenti del sistema a causa dei fault.

2.1 Manuale utente

Per configurare i parametri iniziali del sistema, è necessario modificare il file *config.toml*, nel quale è possibile dare il valore desiderato ai seguenti parametri:

- `num_elements:` numero di elementi randomici contenuti nell'array sul quale viene effettuato il Bubble Sort.
- `num_faults:` numero di fault iniettati nel sistema, ai quali verrà assegnato in maniera randomica l'indice dell'array da manipolare.
- `max_bit_to_flip:` limite superiore per la posizione del bit da flippare, che verrà selezionata in modo casuale fino a questo valore massimo (escluso).
- `fault_injection_time:` limite superiore per il tempo di iniezione del fault (ms), che verrà scelto in modo casuale entro questo intervallo massimo.

2.2 Manuale programmatore

Il progetto è strutturato nei seguenti moduli:

Redundant.rs

Contiene la struttura dati `Redundant`, che rappresenta una variabile ridondante.

- **`Redundant`**:
  - Attributi:
    - `first`: valore originale.
    - `second`: valore duplicato.
  - Metodi:
    - **`new(first: T)`:** Crea una nuova variabile ridondante con due copie identiche.
    - **`is_valid()`:** Controlla se `first` e `second` sono congruenti.
    - **`get()`:** Restituisce il valore se valido, altrimenti genera un errore.
    - **`set(new_value: T)`:** Aggiorna entrambe le copie con un nuovo valore.

Sorting.rs

Implementa tre versioni dell'algoritmo Bubble Sort, utilizzate per stimare l'overhead computazionale dovuto alla ridondanza:

- **`bubble_sort()`**: versione concorrente che opera su un array condiviso tra thread, sincronizzando l'accesso ai dati con un mutex.
- **`redundant_bubble_sort()`**: esegue il Bubble Sort su un array di variabili ridondate, utilizzata per misurare l'impatto della ridondanza sul tempo di esecuzione.
- **`non_redundant_bubble_sort()`**: Bubble Sort su un array normale, utilizzato come riferimento per confrontare le prestazioni con la versione ridondata.

Fault.rs

Contiene la struttura dati `Fault`, che rappresenta un guasto, e i metodi per la relativa generazione e iniezione.

- **`Fault`**: Struttura che rappresenta un guasto.
  - Attributi:
    - `index`: indice della variabile da colpire.
    - `bit_to_flip`: posizione del bit da invertire.

        - `time`: ritardo prima dell'iniezione.
- `generate_faults()`: Crea una lista di guasti casuali basati sui parametri configurati.
- `inject_fault()`: Inverte il bit specificato nella variabile ridondante specificata
- `fault_injector()`: Inietta un guasto su un array condiviso, sincronizzandosi con gli altri thread e gestendo il ritardo specifico del fault.

Analyzer.rs

Consente di analizzare e registrare i risultati del sistema. Contiene la struttura dell'Analyzer e i metodi per il calcolo dell'overhead.

- `Analyzer`:
    - Attributi:
        - `fault_counts`: numero di esecuzioni in cui si è verificato un errore nell'esecuzione del bubble sort.
        - `correct_runs`: numero di esecuzioni in cui il bubble sort ha correttamente ordinato l'array.
        - `incorrect_runs`: numero di esecuzioni in cui il bubble sort non ha correttamente ordinato l'array..
    - Metodi:
        - `new()`: Crea un nuovo oggetto di tipo Analyzer assegnando a tutti gli attributi valore 0.
        - `log_fault()`: Incrementa di 1 l'attributo "fault_count" e aggiorna il log della rilevazione di un fault.
        - `log_result()`: Registra una esecuzione corretta o errata, incrementando rispettivamente di 1 l'attributo "correct_runs" o "incorrect_runs".
        - `report()`: Stampa il report finale, ovvero il valore di tutti gli attributi dell'Analyzer.
        - `report_to_file()`: Crea un file sul quale scrive il report finale, ovvero il valore di tutti gli attributi dell'Analyzer.
        - `generate_report()`: Formatta in tipo stringa il report dell'Analyzer

- `measure_memory_overhead()`: Scrive sul file "memory_overhead_report.txt" le metriche relative all'overhead in memoria dell'array ridondato
- `measure_cpu_time_overhead()`: Scrive sul file "cpu_time_overhead_report.txt" le metriche relative all'overhead nel tempo dell'ordinamento dell'array ridondato

---

Utility.rs

Contiene la struttura dati `Config`, che contiene

- **`Config`**: Struttura che contiene i parametri di avvio del programma.
    - Attributi:
        - `num_elements`: numero degli elementi dell'array da ordinare.
        - `num_faults`: numero dei guasti da iniettare.
        - `max_bit_to_flip`: posizione massima del bit su cui eseguire il flip.
        - `fault_injection_time`: massimo ritardo in ms prima dell'iniezione del guasto.
- **`load_config()`**: Estrae i parametri da un file input in formato toml e li inserisce nella struttura dati "Config"
- **`generate_random_array()`**: Genera un array casuale di variabili ridondanti

Main.rs

Il file principale che coordina l'intero programma, mediante le seguenti fasi:

1. **Inizializzazione**
    - Caricamento dei parametri da un file di configurazione (`config.toml`), come:
        - Numero di elementi da ordinare.
        - Numero di guasti da generare.
        - Limite di bit da invertire.
    - Inizializzazione dell'istanza di un Analyzer per registrare l'analisi.
2. **Misurazione separata dell'Overhead**
    - Calcolo e registrazione di:
        - Overhead di memoria.
        - Overhead temporale della CPU.
3. **Generazione e Iniezione di Guasti**
    - Generazione di una lista di guasti casuali tramite `generate_faults()`.
    - Ogni guasto viene applicato su un array condiviso utilizzando thread separati.
4. **Esecuzione dell'Ordinamento**
    - Sincronizzazione garantita tramite barriere (`Barrier`).
    - Esecuzione dell'iniezione di un guasto in contemporanea all'esecuzione del **Bubble Sort** sull'array.
    - Controllo della correttezza del risultato:
        - Se i valori dell'array sono ordinati e congruenti, il sorting è considerato corretto.
        - In caso di errori o incongruenze, registra l'errore.
5. **Analisi dei Risultati**
    - Utilizzo di `Analyzer` per:
        - Registrare il numero di guasti rilevati e il risultato delle esecuzioni.
        - Creare un rapporto dettagliato sia in console che su file.

3. Risultati

Abbiamo analizzato il comportamento del sistema eseguendo **20 run** con la seguente configurazione iniziale:

- **num_elements** = 500
- **num_faults** = 5000
- **max_bit_to_flip** = 8
- **fault_injection_time** = 10 ms

I valori medi ottenuti sono:

- **Fault rilevati**: 4760 ± 13 (95.2%)
- **Esecuzioni corrette**: 11 ± 3 (0.2%)
- **Esecuzioni invalide**: 229 ± 14 (4.6%)

Distinguiamo tre possibili esiti per ciascuna esecuzione:

- Fault rilevati: il sistema ha identificato una modifica nei dati e ha segnalato l'errore;
- Esecuzioni corrette: il sorting è stato completato senza errori, nonostante l'iniezione dei fault, suggerendo che le alterazioni introdotte non abbiano avuto impatto sul risultato finale (l'ordine è stato mantenuto);
- Esecuzioni invalide: il sorting si è concluso ma il risultato non è ordinato, quindi i fault sono stati iniettati nell'array su elementi già analizzati e ordinati.

Sono state inoltre testate altre configurazioni variando il numero di elementi dell'array e il numero di fault iniettati. I risultati evidenziano che il comportamento del sistema dipende fortemente dalla scelta dei parametri. In particolare, con array di dimensioni ridotte, il sorting viene completato più rapidamente, quindi potrebbe essere necessario un tempo di injection dei fault più basso per poter avere effetto.

In fase di debugging, è stato verificato che la classificazione delle esecuzioni nelle tre categorie fosse corretta, controllando che i fault rilevati corrispondessero effettivamente a errori segnalati dal sistema, che le esecuzioni corrette non avessero subito alterazioni nell'ordinamento e che le esecuzioni invalide presentassero errori coerenti con le alterazioni introdotte dei fault.

Impatto sulla memoria:

L'overhead di memoria derivante da questa tecnica è stato quantificato come segue:

- **Dimensione di un i32**: 4 byte
- **Dimensione di un Redundant<i32>**: 8 byte
- **Overhead di memoria per elemento**: 4 byte
- **Memoria totale senza ridondanza (500 elementi)**: 2000 byte
- **Memoria totale con ridondanza (500 elementi)**: 4000 byte
- **Overhead totale di memoria (500 elementi)**: 2000 byte

L'introduzione della ridondanza raddoppia l'uso della memoria, passando da 2000 a 4000 byte per 500 elementi, questo è coerente con la definizione di <Redundant<T>> che

contiene due copie dello stesso valore, quindi la sua dimensione sarà il doppio della dimensione del tipo <T>.
L'occupazione di memoria è stata verificata sperimentalmente utilizzando la funzione std::mem::sizeof::<T>().

Impatto sulla CPU:

Per quanto riguarda i tempi di CPU, sono stati ottenuti i seguenti valori medi, calcolati su array di 10 000 elementi piuttosto che 500, per evitare risultati nulli (con 500 si otterrebbero valori al di sotto del ns) e garantire misurazioni comparabili:

- **CPU time with redundancy:** 892.32 ± 34.47 ms
- **CPU time without redundancy:** 444.83 ± 34.90 ms
- **CPU time overhead:** 444.83 ± 6.81 ms

L'overhead computazionale per 500 elementi dovuto alla ridondanza è in media circa 444.83 ms. Questo incremento è coerente con l'aumento del numero di operazioni dovute alla gestione della ridondanza, che include il mantenimento di due copie dei dati e le verifiche di congruenza.

4. Conclusioni

La tecnica di iniezione di guasti bit-flip ha permesso di valutare la robustezza di un sistema che utilizza ridondanza dei dati in Rust. I risultati indicano che l'approccio ridondante rileva efficacemente gran parte delle alterazioni indotte, ma comporta un raddoppio dell'uso di memoria e un incremento del tempo di esecuzione rispetto alla versione non ridondante. Pertanto, la scelta di adottare una struttura dati duplicata va bilanciata tra le esigenze di affidabilità e le risorse di calcolo e memoria disponibili.