POLITECNICO DI TORINO

Corso di Laurea Magistrale

In INGEGNERIA INFORMATICA

Tesi di Laurea (prev. D.M 509/99)

Modello predittivo e proposta metodologica per la stima dell'uso della CPU in sistemi infotainment multi-core

Relatore

prof. Massimo VIOLANTE

Candidato

Paolo MASSIMINO

Anno Accademico 2018/2019

A mia moglie

Indice

Capitolo 1

Il problema e l'approccio proposto

1.1 Sistemi *infotainment* in ambiente *automotive*

Negli ultimi anni gli autoveicoli hanno sperimentato un incremento di tecnologia on board senza precedenti nella storia dell'auto.
Funzionalità avanzate come i sensori di parcheggio, la frenata assistita o la telecamera posteriore sono ormai offerti dai *car maker* ai loro clienti come elementi di base dell'allestimento dell'auto. Anche altri dispositivi, relativamente semplici come le smart keys o più complessi come gli head-up display sono diventati, a seconda della fascia di mercato al quale il veicolo è destinato, da elementi *optional* a immancabili presenze stabili a disposizione del guidatore.

Un elemento che è diventato sempre più un fattore discriminante per la scelta di uno specifico modello di auto da parte dell'acquirente, alla stregua delle caratteristiche motoristiche e di design, è il sistema integrato *infotainment*.
Con la parola infotainment, si intende una serie di funzionalità multimediali: dal comandare dall'abitacolo della vettura i sistemi audio (radio e riproduzione musicale), alla disponibilità di utilizzare ed istruire il navigatore satellitare, alla gestione della telefonata in ingresso o in uscita tramite dispositivi *handsfree buetooth*.
Queste funzioni sono implementate su un singolo sistema installato nell'auto, che prende il nome, appunto, di sistema infotainment.

Il sistema infotainment è generalmente composto da almeno i seguenti elementi:

- display centrale con interfaccia uomo-macchina (HMI, *human-machine interface*), spesso in configurazione *touch*, cioè sensibile al tocco: è il principale veicolo di ingresso e uscita di informazioni da e verso il guidatore e il passeggero;
- Sistema di navigazione satellitare: si tratta del software e dell'hardware necessario a ricevere il segnale GPS e ad utilizzarlo per definire su una mappa la posizione del veicolo e a mostrarla sul display dell'infotainment; permette inoltre di pianificare un percorso stradale e di ricevere le indicazioni in tempo reale (turn-by-turn) da seguire per raggiungere la destinazione;
- Diffusori audio integrati nella plancia dell'autovettura; in numero variabile da 2 a n, sono utilizzati per diffondere il segnale audio all' interno dell'abitacolo;
- Connessione per dispositivi USB; permette il collegamento di comuni *device* di *storage* contenenti file multimediali, come audio MP3 o altri formati;
- Connettività *bluetooth*, utilizzata per collegare senza fili uno smartphone al sistema di infotainment, implementando i profili handsfree per chiamate viva voce in auto, streaming audio dallo smartphone al sottosistema audio, etc…
- Controlli a riconoscimento vocale: oltre all'interfaccia presente sul display, il guidatore o il passeggero possono impartire comandi al sistema infotainment tramite la voce; un sistema di riconoscimento vocale presente nel sistema convertirà i comandi vocali in azioni sul sistema, come il cambio di stazione

radio, o della sorgente audio al momento utilizzata, o all' inserimento dell'indirizzo stradale per il navigatore GPS, etc.;

- Funzionalità di *mirroring* del contenuto dello smartphone, ad esempio i noti CarPlay di Apple e Android Auto d Google.

Altre funzionalità disponibili sui sistemi di fascia alta possono prevedere la connettività tramite WiFi, players video HD, microfoni direzionali, *rear seat entertainment* (display aggiuntivi, solitamente inseriti sul retro dei poggiatesta anteriori per permettere l'interazione con il sistema infotainment anche ai passeggeri seduti sui sedili posteriori), connettività infrarosso per cuffie audio, Head up display anteriore, etc…

Le funzionalità dell'infotainment sono implementate all'interno una singola *silver box*, la quale deve essere equipaggiata di hardware adeguato a supportare l'insieme delle applicazioni, come un SoC (system-on-chip) sufficientemente potente in termini sia di carico computazionale, sia di capacità grafiche, sia di quantità e tipologia di memoria RAM e di storage.

Il progetto del sistema deve anche tenere in conto i requisiti richiesti per poter produrre un prodotto in grado di essere utilizzabile in ambito *automotive*; come ad esempio il range di temperatura entro il quale il sistema nel suo complesso deve essere in grado di funzionare, le vibrazioni e le accelerazioni alle quali il sistema sarà sottoposto, l'*aging* da sopportare (tipicamente 20 anni dall'uscita dall'impianto di fabbricazione).

1.2 Convergenza di funzionalità in una singola centralina

Negli ultimi anni, l'organizzazione delle funzionalità presenti in un'auto ha subito una radicale trasformazione. Funzioni che generalmente sono implementate da centraline indipendenti tendono ad essere sempre di più aggregate in una singola centralina: questa strategia ha visto coinvolti elementi importanti come l'infotainment stesso e l'*instrument cluster* (il quadro di bordo con gli strumenti che

si trovano tradizionalmente dietro al volante). La coesistenza di queste due funzionalità in una sola "*silverbox*" ha posto nuove sfide ai progettisti, sia in termini di organizzazione e partizionamento dell'hardware e dei display, sia in termini di architetture software e utilizzo delle risorse computazionali e grafiche disponibili sul system-on-chip scelto.

Ulteriori funzionalità, classificabili nelle categorie dei così detti ADAS ("*Advanced Driver Assistance Systems*" – Sistemi avanzati per l'assistenza alla guida), tradizionalmente offerti come centraline separate, possono trovare vantaggi sia economici, sia di performance, nell'integrazione su singolo sistema.

Alcuni esempi di sistemi ADAS sono:

- Adaptive Cruise Control,
- Blind Spot Assistant,
- Foreward Collision Warning
- Park Assistant,
- Traffic Sign Recognition
- Lane Keeping System
- Driver Monitoring System
- Autonomous Emergency Braking
- Adaptive Light Control

Essi possono essere annoverati tra i sistemi di base necessari per poter iniziare a definire il futuro veicolo autonomo, al netto delle funzionalità di intelligenza artificiale ancora da sviluppare per poter governare un veicolo tramite sistemi totalmente automatici.

Ad oggi pertanto il trend è chiaro: un "sistema veicolo" multi centralina, è quindi intrinsecamente distribuito, si trasforma sempre di più in un sistema più centralizzato, equipaggiato pertanto con un numero sempre crescente di unità di esecuzione (cores) all'interno della CPU.

1.3 Dimensionamento delle risorse di sistema e scelta del SoC

Una delle sfide che l'architetto di sistemi di infotainment deve fronteggiare è il corretto dimensionamento delle capacità computazionali, e quindi della scelta del fornitore e dello specifico part number, del system-on-chip intorno al quale ruota l'intera architettura di sistema.
Da punto di vista dell'architettura software, i vincoli principali da considerare per il "*best fit*" sono:

- garantire la necessaria potenza di calcolo alle applicazioni che, per loro natura, richiedono di essere eseguite per tutto il ciclo vita della sessione
- riservare sufficiente potenza di calcolo per i picchi di carico computazionale dovuti all'attivazione di applicazioni on demand da parte dell'utilizzatore
- massimizzare il riuso di codice legacy provenienti da altri progetti simili, valorizzando gli asset aziendali sviluppato nel corso del tempo
- garantire la coesistenza di applicazioni critiche sotto il profilo di sicurezza (nell'accezione del termine inglese "*safety*", cioè indirizzato alla salvaguardia dell'integrità fisica dell'occupante) e applicazioni non *safety-related*

L'ottimizzazione delle risorse di sistema, vincolata ai fattori precedentemente esposti, è pertanto un problema di difficile soluzione e di altrettanta difficile dimostrazione: la stima dell'utilizzo reale delle risorse di sistema per un nuovo prodotto in fase di definizione viene generalmente espletata tramite:

- conoscenze parziali del codice delle applicazioni reali, dovuto al fatto che spesso alcuni fornitori non rilasciano il codice sorgente di alcune librerie ma soltanto il codice binario compilato per proteggere la proprietà intellettuale;
- impiego di dati storici non sempre facilmente confrontabili con il nuovo scenario

I dati a disposizione vengono quindi catalogati per generare un insieme di funzioni di sistema, ognuna delle quali sarà sorgente di richieste computazionali.

Tramite la definizione di alcuni use cases, si combinano le funzioni sommando le loro richieste computazionali, per ottenere una stima delle risorse utilizzate.
Questo approccio si è spesso rivelato poco aderente al reale fabbisogno di risorse, sia in difetto, sia in eccesso, con la conseguente sottoutilizzazione o sovraccarico del sistema hardware prescelto.

È evidente che una non corretta stima delle necessità computazionali si ripercuote, in ultima analisi, in un costo aggiuntivo spesso ingente:

- in caso di sovrastima, il costo sostenuto per l'hardware aggiuntivo e, di riflesso la perdita di competitività del prodotto in termini di costo finale al cliente con, al limite, il rischio di perdere la commessa in favore di un *competitor* in grado di proporre al cliente un prodotto a minor costo a parità di funzioni offerte;
- in caso di sottostima, il costo sostenuto per una successiva riprogettazione dell'hardware, a cui vanno aggiunti i costi relativi alla gestione del fornitore, ritardi nella messa in produzione del sistema, eventuali penali, etc...

Lo scopo di questo lavoro è la proposta e lo studio di un modello che permetta una stima più accurata delle risorse necessarie a soddisfare le richieste di un sistema infotainment - in particolare per quanto riguarda l'impiego della CPU e dei cores disponibili - per un corretto dimensionamento e, di conseguenza, una corretta scelta, del system on chip utilizzato.

Infatti, un core aggiuntivo presente sul system-on-chip può dare benefici in termini prestazionali, a fronte di un costo aggiuntivo, ma solo se l'intero impianto software è in grado di utilizzarlo almeno per una percentuale di tempo adeguata: se questa condizione non risultasse vera, l'aggiunta di un core addizionale risulterebbe, al contrario, uno svantaggio competitivo, ottenendo pertanto un effetto opposto a quello voluto.

Le attività svolte durante questo studio sono le seguenti:

- definire un modello che permetta di predire la capacità di un set di applicazioni di utilizzare l'eventuale incremento di uno o più cores addizionali;
- analizzare use cases reali in ambito infotainment utilizzando piattaforme di esecuzione e suites software esistenti, raccogliendo dati sul campo che permettano di confrontare i risultati del modello con risultati misurati;
- applicare il modello ai dati accolti per identificare gli incrementi di prestazioni attesi al variare del numero di cores addizionali e confrontare le stime del modello con in valori prestazionali misurati;
- definire un sistema di simulazione in grado di sostituire, in alcuni contesti, la misura stessa dei dati reali sul campo, attraverso la generazione di "carichi computazionali" statisticamente aderenti ai comportamenti delle applicazioni reali.

Capitolo 2

Lo scheduling dei processi

2.1 Processi e Threads

Storicamente, in ambito informatico, una istanza di un programma in esecuzione ben definita dal punto di vista del codice eseguibile, il quale pertanto resta immutato per tutta la durata dell'esecuzione stessa, prende il nome di processo.
Un processo, nei sistemi moderni, viene eseguito sotto il controllo di un altro processo speciale e, generalmente dotato di privilegi superiori: il sistema operativo.
Un singolo processo non è definito solamente dal codice eseguibile che lo compone, ma anche da una serie di informazioni variabili durante la sua esecuzione: lo stato dei registri del microprocessore che lo esegue, i thread dipendenti e creati dallo stesso processo, tutti i descrittori dei file, le aree di *shared memory* condivise con altri

processi o threads, le periferiche attualmente in uso, dai dati temporanei presenti nello heap, e così via.
Un processo è anche da considerare come la punta della piramide di dipendenze composta dagli eventuali thread creati da esso stesso: un singolo thread diviene quindi l'unità atomica sequenziale minima nella quale un processo può essere suddiviso. Ogni thread può quindi essere eseguito tramite la supervisione del sistema operativo in serie ad altri thread, nel caso il sistema hardware disponga di una sola unità di esecuzione, sia in concorrenza con altri thread, nel caso in cui più unità di esecuzione, o cores, siano disponibili.

Il singolo thread viene eseguito in maniera indipendente dallo stesso processo che lo ha generato, il quale può, tramite meccanismi specifici messi a disposizione dal sistema operativo, condividere con esso parti della memoria, in lettura e/o in scrittura. Ogni thread, per poter essere eseguito indipendentemente dal processo generante, sarà anche provvisto di uno stack dedicato.

Il significato della parola inglese "thread" suggerisce l'immagine dei fili rappresentati dai vari thread per comporre l'intera trama del processo in esecuzione.
Dal momento che il thread è l'unità minima di esecuzione seriale di un processo, ogni processo deve possedere almeno un thread, cioè sé stesso, o un numero variabile generalmente limitato superiormente dalle strutture dati del sistema operativo predisposte per tenere traccia di ogni thread eseguito sul sistema.
Questa quantità può essere configurata, ad esempio su sistemi Linux, agendo su dedicate variabili di sistema, a run time, cioè senza la necessità di ricompilare il kernel, ma sempre con un limite massimo determinato al momento della compilazione del kernel stesso.

Esiste tuttavia una differenza sostanziale tra la creazione del primo thread, cioè del processo stesso, e la creazione dei thread successivi: alla creazione del primo thread (il processo stesso) il sistema operativo deve assegnare tutte le risorse richieste per la sua corretta esecuzione, deve essere caricato in memoria il codice da eseguire, devono

essere risolte eventuali dipendenze da librerie di sistema o di utente, devono essere creai i riferimenti ai files, etc.
Queste operazioni sono generalmente onerose e introducono quindi un tempo di *startup* del processo prima che la prima istruzione presente nel programma possa essere eseguita dal core.

Essendo invece il thread (successivo) già parte del processo, e condividendone il codice eseguibile e i dati in memoria, richiede un tempo di attivazione che sarà generalmente inferiore anche di un ordine di grandezza o più.
Ciò suggerisce che una suddivisione di un processo in un numero di thread separati permette di allocare una sola volta alcune risorse critiche comuni a tutti i thread e instaurare dei meccanismi di comunicazione e sincronizzazione tramite la memoria condivisa accessibile da tutti i thread e da meccanismi di lock adeguatamente disegnati per evitare il rischio di dead locks.

Ma quanti thread in generale possono essere eseguiti contemporaneamente su un sistema?
Un sistema operativo multitasking è in grado di eseguire contemporaneamente un numero di thread pari al numero di unità di esecuzione presenti nel sistema. Questo significa che solamente un numero limitato di thread può avere il pieno controllo dei cores disponibili in un dato momento temporale.
Per poter permettere a tutti i thread di accedere alle unità di esecuzione, è necessario definire delle politiche di condivisione di tempo di esecuzione, in modo da allocare ad ogni thread, secondo il proprio turno, una unità di tempo stabilita per poter essere eseguiti.

Superato questo tempo limite, il thread sarà messo in pausa e richiamato in un secondo tempo secondo l'algoritmo di scheduling prescelto, in modo da dare all'utente l'impressione che ogni thread venga eseguito in contemporanea sul sistema. Questo scambio di thread in esecuzione prende il nome di cambio di contesto o *context switch*.

A questo fine, l'unità di tempo, o *timeslice*, che il sistema operativo assegna al thread per poter essere eseguito non può essere troppo elevata, se si vuole dare l'impressione all'utente di utilizzare un sistema sufficientemente reattivo ai comandi.
Il *timeslice* non può, d'altro canto, nemmeno essere troppo breve, poiché il processo di context switch è oneroso dal punto di vista temporale: l'overhead necessario ad un context switch deve quindi essere il più possibile trascurabile rispetto all'effettivo tempo di esecuzione del thread durante il suo timeslice, in modo da evitare che il sistema passi la maggior parte del tempo ad eseguire context switch invece di eseguire il codice dei threads.

Il corretto dimensionamento e ottimizzazione della durata del timeslice è un problema di non facile soluzione in caso di sistemi operativi progettati per un utilizzo così detto *general purpose* a causa della varietà pressoché imprevedibile della tipologia e del numero di threads che concorrono all'uso delle unità di esecuzione.

In sistemi più controllati, come ad esempio alcuni sistemi embedded *real time*, la variabilità è maggiormente prevedibile e il dimensionamento del timeslice può essere affrontato più agevolmente.
In generale, il sistema operativo Linux definisce un *timeslice* minimo di 4ms.

Un thread può subire un cambio di contesto anche se il timeslice a lui assegnato non è ancora scaduto. In generale questo evento accade a fronte di una chiamata di sistema o ad un'attesa da parte del thread della esecuzione di una attività di input/output, compresi lock o meccanismi di sincronizzazione.

Questo tipo di context switch prende il nome di *Voluntary Context Switch*, poiché può essere in qualche modo considerato "volontario" da parte del thread: esso è legato alla necessità del thread di attendere che un evento abbia seguito.
Questo tipo di context switch si contrappone al precedente tipo di context switch, causato dall'esaurirsi del timeslice, il quale prende il nome di *Unvoluntary Context Switch*.

2.2 Stati di un processo

Di seguito sono elencati gli stati ai quali si può trovare un thread/processo attivo. Dall'elenco vengono esclusi gli stati *zombie*, non di interesse al momento.

Un thread può essere:

- In esecuzione o *running*: al processo è associato un core per la durata del suo timeslice ed esso viene eseguito normalmente;
- pronto, o *ready*: il processo non è in esecuzione, bensì posto in una coda di processi in attesa gestita dallo scheduler, il quale, secondo la sua politica di gestione delle code di attesa, si occuperà di effettuare il context switch a tempo debito;
- sospeso, o *suspended*: il processo non è in esecuzione, ma è posto in una coda che contiene i processi che hanno eseguito una chiamata di sistema. Quando la chiamata di sistema avrà seguito, il processo sarà spostato nella coda dei processi *ready*, in attesa di essere messo in esecuzione. La chiamata di sistema può essere una richiesta di allocazione di memoria, o una primitiva di sincronizzazione, o un accesso ai dispositivi di I/O, etc.

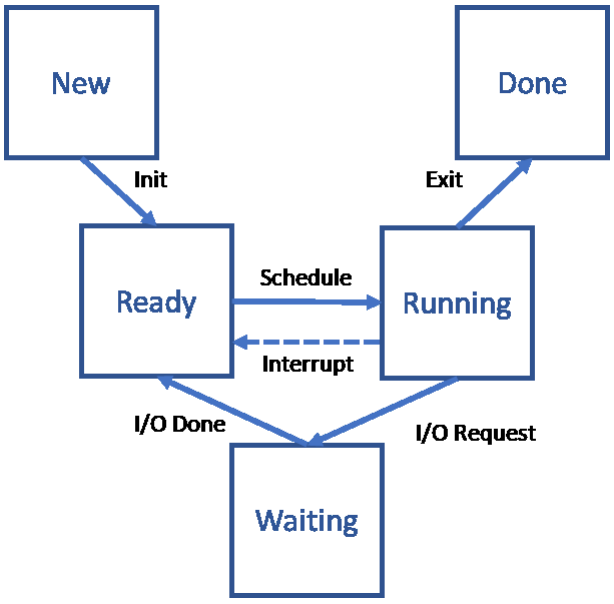

Figure 1 - Diagramma di stato di un processo

Come precedentemente accennato, quando una unità di esecuzione viene associata dallo scheduler ad un altro processo in stato di ready e contestualmente il processo precedentemente in esecuzione viene posto nella coda di ready o di suspended, avviene il così detto context switch.
Un context switch prevede il salvataggio di tutte le informazioni necessarie affinché sia possibile ripristinare l'esecuzione del processo come se il context switch non fosse avvenuto.

I dati minimi necessari per un ripristino ordinato prevedono il *program counter* e i registri di stato della CPU, i vari flags, l'identificatore della CPU sulla quale il processo era in esecuzione, puntatore alla page table, le informazioni legate al memory management, etc.

Nei sistemi operativi multitasking, esiste una struttura dati dedicata allo scopo, la quale prende il nome di *Process Control Block* (PCB) del processo.

Il passo successivo al context switch, è la scelta del processo da eseguire da parte dello scheduler. La scelta avviene tra i processi presenti nella coda dei processi "ready" e segue una predefinita politica di scheduling. Esso accederà al proprio PCB per il necessario ripristino delle condizioni precedenti al context switch. Una delle politiche di scelta del processo da eseguire, ampiamente utilizzate nei sistemi Linux, è chiamata *Completely Fair Scheduler* (CFS).

Il context switch causerà anche una invalidazione della cache della CPU: ciò per assicurare che ogni processo possa accedere solamente ai propri dati. Questo passo del processo di context switch è spesso considerato uno dei più onerosi per quanto riguarda l'impatto sulle prestazioni.

2.3 Classificazione dei processi

Un processo, tradizionalmente, può essere classificato, secondo il suo profilo d'uso delle periferiche o della CPU, in due diverse categorie:

- I/O-Bound
- CPU-Bound

Al primo tipo vengono annoverati i processi che richiedono un intenso utilizzo dei dispositivi di ingresso/uscita, o interagiscono particolarmente con il sistema operativo, restando in attesa che il dispositivo risponda o che la chiamata di sistema abbia termine.
Il secondo tipo di processo utilizza maggiormente le capacità computazionali del sistema, eseguendo per più tempo consecutivo parti di programma che non necessitano di dati presenti sui dispositivi di I/O o non richiedono di interagire con le chiamate di sistema.

In generale i processi possono avere sia una componente I/O-bound, sia una componente CPU-Bound: la classificazione quindi è spesso sfumata, e a determinare la tipologia di processo è per lo più il suo comportamento prevalente.

Esistono altre possibili classificazioni ai quali i processi possono essere soggetti, se analizzati dal punto di vista della interazione con altre parti del sistema, o con l'utilizzatore dello stesso:

- *batch*
- periodici
- *interactive*
- *event triggered*

Il primo tipo di processo non necessità di interagire con l'utilizzatore del sistema: la sua esecuzione può essere pianificata indipendentemente da eventi esterni e ripetuta senza modifiche sostanziali al flusso di esecuzione. Questo tipo di processo può essere eseguito in background ad altri processi, e la sua priorità può essere impostata ad un livello inferiore rispetto alle altre tipologie.

Il gruppo dei thread periodici può essere considerato un sottoinsieme dei processi di tipo batch: generalmente la loro esecuzione è controllata da un timer.

I processi *interactive* sono una classe di processi che interagiscono con l'utente e necessitano generalmente, per garantire un tempo di reazione minimo alle azioni dell'utente, una priorità sufficientemente alta da non essere interrotto da altri processi, ad esempio quelli di tipo batch.

I processi *event triggered* ricadono nei processi asincroni che rispondono ad eventi di sistema o ricezione di messaggi. I processi che gestiscono interrupt, ad esempio causati dalla pressione di un tasto della tastiera o dall'arrivo di un segnale dal DMA (*Direct Memory Access*), ricadono in questa classe.

Come accennato precedentemente, in presenza di una eterogeneità di tipologie di processi concorrenti sullo stesso sistema, la definizione di una strategia ottima di scheduling è un problema arduo da risolvere. Alcune soluzioni ad hoc sono state studiate per ottimizzare specifici casi particolare di ristrette tipologie di processo coesistenti. Il caso generale, nella sua complessità, è tuttora area di studio.

2.4 Concetto di Multitasking

Linux è un sistema operativo con supporto al multitasking: cioè offre la capacità di eseguire più processi contemporaneamente arbitrando l'utilizzo della unità di elaborazione attraverso la gestione di code di attesa e assegnamento di determinati

timeslice di esecuzione ad ogni processo. La scelta di scambiare due processi, il primo in esecuzione, il secondo nella coda dei processi *ready*, viene demandata ad un componente del sistema operativo che prende il nome di *scheduler*. Il vero e proprio cambio di contesto viene però effettuato da un secondo componente del sistema operativo: il *dispatcher*, il quale si occupa di gestire materialmente il context switch, salvando e ripristinando le strutture dati e i registri fondamentali della CPU per permettere al processo subentrante di poter riprendere l'esecuzione. La ripartizione è definita tramite i vari algoritmi di scheduling, i più noti in letteratura sono elencati nel paragrafo 2.5.

La capacità di un sistema operativo di supportare il multitasking può essere raggiunta attraverso due paradigmi fondamentali:

- attraverso un sistema cooperativo (*cooperative*)
- attraverso prelazione della CPU (sistema *preentive*)

Il primo paradigma prevede l'impossibilità da parte del sistema operativo di interrompere volontariamente il processo in esecuzione: sarà esso stesso, secondo strategie interne proprie, o a fronte di richieste di accesso ai servizi di input-output.
Questa strategia delega, di fatto, la percezione verso l'utente nell'esecuzione concorrente di più processi, alla collaborazione reciproca degli stessi: è evidente che, se un processo resta bloccato in un loop infinito senza lasciare il controllo, tutto il sistema risulterà bloccato.

Il secondo paradigma prevede che sia il sistema operativo a interrompere il processo correntemente in esecuzione allo scadere di un determinato periodo di tempo (*timeslice*), o al subentrare di eventi, come interrupt a priorità superiore, interrupt non mascherabili, o la richiesta di eseguire processi a più alta priorità, indipendentemente dal fatto che il *timeslice* concesso al processo in esecuzione sia scaduto o meno.

2.5 Concetto di Scheduling

Come accennato nei paragrafi precedenti, in presenza di sistemi mutitasking di tipo preemptive è necessario poter gestire un gran numero di processi concorrenti, generalmente in numero molto superiore al numero di CPU disponibili, i quali possono essere invocati dall'utente in una qualsiasi sequenza temporale. Ogni processo può poi essere composto da un solo thread o da un numero variabile di essi, alcuni dei quali intrinsecamente CPU-Bound, altri più I/O-Bound, o una via di mezzo.

Per poter gestire correttamente questo scenario, un ruolo fondamentale è ricoperto dallo *scheduler* dei processi, il quale deve essere in grado di garantire che tutti i processi possano accedere alla CPU senza soffrire dell'effetto di *starvation* (letteralmente "inedia"), favorire la reattività del sistema nel suo complesso minimizzando la latenza, massimizzare l'utilizzo della CPU per l'esecuzione "utile" (cioè escludendo i tempi di context switch), completare un processo entro un tempo prestabilito (in sistemi operativi così detti real-time, è uno dei requisiti fondamentali).

Il contesto specifico in cui il sistema è utilizzato, può suggerire la strategia di scheduling più appropriata, in accordo con l'obiettivo che si vuole perseguire: Per i processi di tipo batch si vuole prediligere e massimizzare il throughput, quindi il tempo di esecuzione rispetto al tempo di context switch, mentre in processi più interattivi si predilige la riduzione del tempo di attesa per l'accesso alla CPU, e di conseguenza il tempo medio di reattività.

Gli algoritmi più noti in letteratura sono i seguenti:

- First Come, First Served (FCFS)
- Shortest Job First (SJF)
- Priority scheduling (scheduling per priorità)
- Round Robin (RR)
- Multilevel Queue
- Multilevel Feedback Queue

- Shortest Remaining Time First (SRTF)
- Earliest Deadline First (EDF)
- Rate Monotonic (RM)
- Completely Fair Scheduler (CFS)

Ogni algoritmo possiede le proprie peculiarità e, in generale, la strategia che implementa cerca di affrontare casi specifici di tipologie di processi che concorrono all'utilizzo della CPU. Ad esempio l'algoritmo FCF prevede la gestione dei processi tramite una singola coda FIFO senza *preemption* e gestione delle priorità, applicando pertanto l'approccio più semplificato.
L'algoritmo CFS è la strategia adottata dal kernel di Linux.

Capitolo 3

Il modello predittivo

3.1 Thread Parallelization e System Overbooking Ratios

Nei sistemi multi-utente e multi processo, Il kernel passa da un thread all'altro nel tentativo di condividere la CPU in modo efficace; questa attività, come introdotto nel Capitolo 2 si chiama cambio di contesto (Context Switch). Quando un thread viene eseguito per la durata del suo intervallo di tempo o quando si blocca perché richiede una risorsa che non è attualmente disponibile, il kernel, tramite lo scheduler e le politiche associate, trova un altro thread da eseguire e il contesto passa ad esso. Il sistema può anche interrompere il thread attualmente in esecuzione per eseguire un thread attivato da un evento asincrono, come un interrupt da un dispositivo di I/O.
Sebbene entrambi gli scenari comportino la commutazione del contesto di esecuzione della CPU, la commutazione tra i thread avviene generalmente in modo sincrono rispetto al thread attualmente in esecuzione, mentre gli interrupt si verificano in modo

asincrono rispetto al thread corrente. Inoltre, i cambi di contesto tra processi sono classificati come volontari (Voluntary Context Switch) o involontari (Unvoluntary Context Switch). Un cambio di contesto volontario si verifica quando un thread si blocca perché richiede una risorsa che non è disponibile. Un cambio di contesto involontario ha luogo quando un thread viene eseguito per tutta la durata del suo intervallo di tempo o quando il sistema identifica un thread con priorità più alta da eseguire.

Il modello proposto si basa sulla misura di questi due parametri fondamentali dei processi in esecuzione.

Considerando un dato intervallo di tempo ΔT di osservazione, si definiscono:

- U_{CSW} : numero di unvoluntary context switch nell'intervallo di tempo ΔT
- V_{CSW} : numero di voluntary context switch nell'intervallo di tempo ΔT

Si definisce quindi la quantità *Thread Parallelization Ratio* (TPr) come segue:

$$TPr(\Delta T) = \frac{U_{CSW} * G_u}{U_{CSW} * G_u + V_{CSW} * G_v} \qquad \textbf{EQ 1}$$

$$G_u = \frac{1}{\sum U_{CSW}} \qquad \textbf{EQ 2}$$

$$G_v = \frac{1}{\sum V_{CSW}} \qquad \textbf{EQ 3}$$

Un processo con un alto numero di cambi di contesto volontari, può essere considerato un processo *"I/O Bound"*, cioè con una alta propensione alle attività di input/output che non coinvolgono la CPU.

Parimenti, un processo con un alto numero di cambi di contesto involontari, può essere classificato come processo *"CPU-Bound"*, cioè sarà più propenso ad utilizzare la CPU come risorsa principale.

Il grado di "CPU-Bound" di un processo indica che la velocità con cui il processo avanza è limitata dalla velocità della CPU. il grado di "I/O Bound" indica che la velocità con cui un processo avanza è limitata dalla velocità del sottosistema I/O.

TPr può essere considerato un indicatore del grado di CPU-Boundness o I/O-Boundness di uno specifico processo.

Infatti:

$$\lim_{U_{CSW}\to\infty} TPr(\Delta T) = \lim_{U_{CSW}\to\infty} \frac{U_{CSW} * G_u}{U_{CSW} * G_u + V_{CSW} * G_v} = 1 \qquad \textbf{EQ 4}$$

$$\lim_{V_{CSW}\to\infty} TPr(\Delta T) = \lim_{V_{CSW}\to\infty} \frac{U_{CSW} * G_u}{U_{CSW} * G_u + V_{CSW} * G_v} = 0 \qquad \textbf{EQ 5}$$

Dal che si deduce che un processo "I/O Bound" avrà un valore di $0 < TPr << 1$ e un processo "CPU-Bound" avrà un valore di $1 > TPr >> 0$.

Il valore di *TPr* varia per ogni singolo thread e dipende dal numero di cores disponibili al momento dell'esecuzione del processo.

Infatti, mentre il numero di voluntary context switch può essere considerato costante nella finestra di osservazione, essendo dipendente dagli eventi legati all'I/O o alle chiamate di sistema, il numero di unvoluntary context switch dipende dalla disponibilità di una CPU libera al momento in cui il periodo di esecuzione assegnato a thread dal sistema operativo scade.

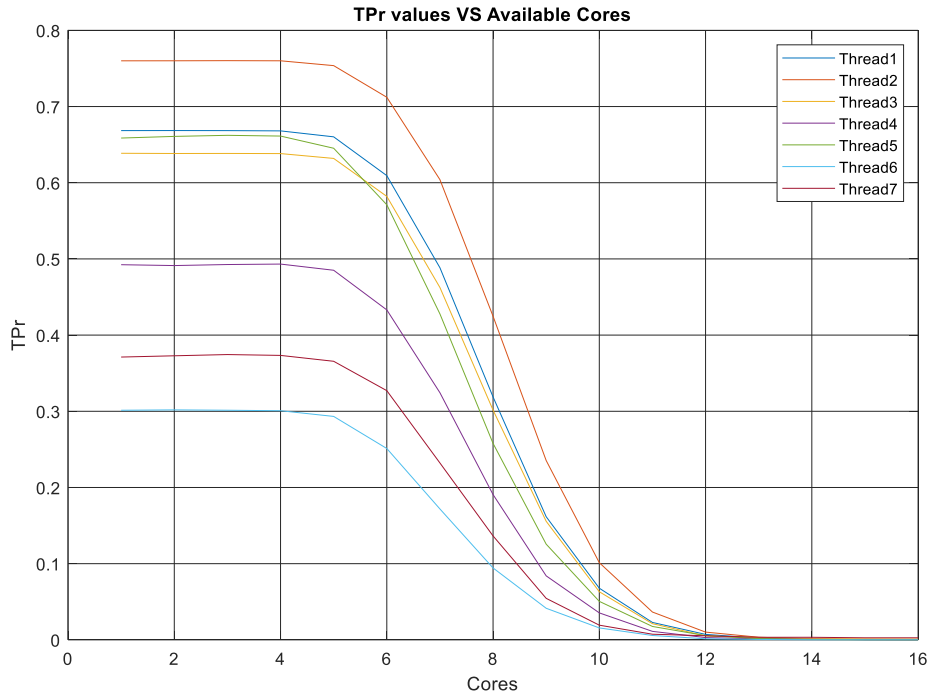

Figure 2 - Andamento di TPr al variare del numero di cores disponibili

In Figure 2 è mostrato l'andamento di *TPr* di alcuni threads in funzione del numero di cores disponibili. Il valore iniziale, riferendosi alla disponibilità di un solo core, rappresenta il valore di CPU-Bound o I/O-Bound del thread considerato.
Un valore di 0.8 è legato ad un alto livello di "CPU-Bound", valori sotto 0.5 sono tipici di threads "I/O-Bound".

All'aumentare del numero di cores disponibili, il numero di unvoluntary context switch si riduce per l'effetto della maggiore disponibilità di CPU ed i valori di *TPr* tendono a 0, in accordo con le formule EQ 4e EQ 5.
Ogni singolo thread ha, come precedentemente introdotto, un proprio valore intrinseco di *TPr*.

Per poter valutale il grado di "CPU-Bound" e "I/O-Bound" dell'intero sistema, si definisca il parametro System Overbooking Ratio come segue:

$$\mathrm{SOr}(\Delta T) = \frac{\sum_{i=1}^{NThreads} TPr(\Delta T, i)}{NThreads} \qquad \textbf{EQ 6}$$

dove:

- *TPr* (ΔT, i) è il Thread Parallelization Ratio del thread i,
- ΔT è la finestra temporale di osservazione
- *NThreads* è il numero di thread in esecuzione concorrente

Il parametro SOr(ΔT) (Figure 3) è limitato superiormente a 1, inferiormente a 0, come conseguenza delle formule EQ 4 e EQ 5.

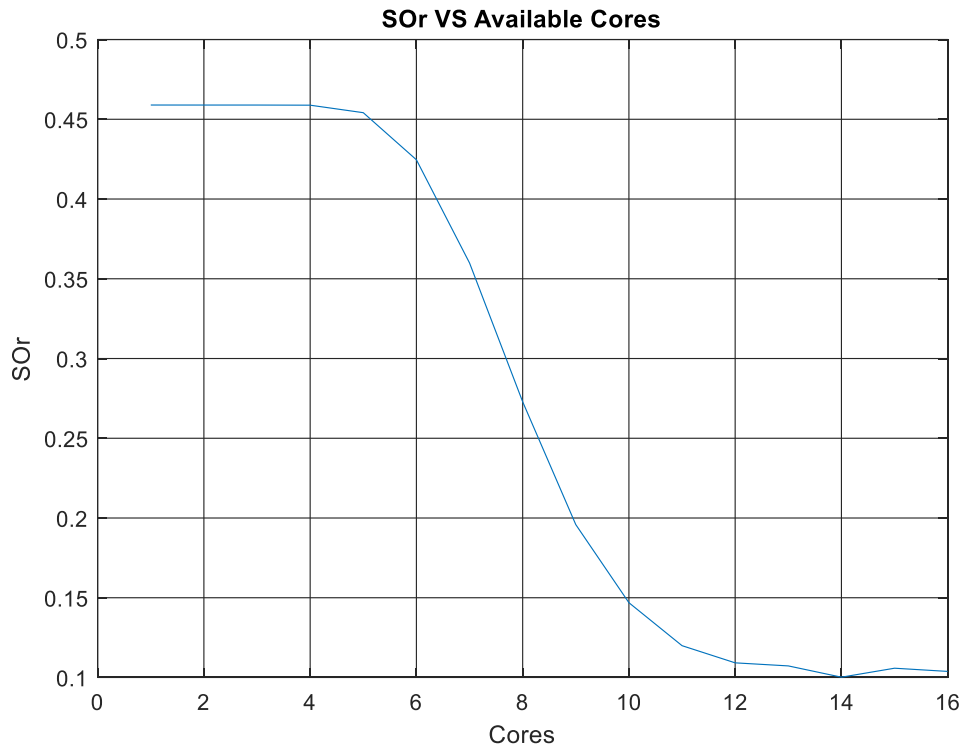

Figure 3 - Andamento del parametro System Overbooking ratio al variare del numero di cores disponibili

I parametri *TPr*(ΔT) e SOr(ΔT), essendo strettamente legati all'andamento del numero di unvoluntary context switch, possono essere quindi utilizzati per predire l'aumento di performance all'aumentare del numero di cores disponibili nel sistema.

3.2 Definizione del modello

La predizione dell'aumento di prestazioni legato ad un core aggiuntivo disponibile durante l'esecuzione del pool di threads, e quindi alla maggiore potenziale parallelizzazione si basa sull'applicazione della legge di Amdahl, i cui dettagli si possono trovare nell'Appendice A.

Richiamando l'espressione della legge di Amdahl

$$S(N) = \frac{1}{(1-p) + \frac{p}{N}} \qquad \textbf{EQ 7}$$

dove:

- S(N): aumento di prestazioni dovute all'aggiunta di N cores
- p: percentuale di applicazione che può essere eseguita in parallelo
- (1-p): percentuale di applicazione che non può essere eseguita in parallelo
- N: Numero di cores aggiunti rispetto al riferimento iniziale

I modello proposto prevede di definire, come valore di *p* la quantità TPr (ΔT, i).

Pertanto il parametro che, nella legge di Amdahl, viene indicato come percentuale di parallelizzazione del singolo processo, viene, nell'approccio proposto, sostituito da un termine legato al numero di unvoluntary context switch, cioè all'indicatore legato all'evento che il processo, se non fosse scaduto il *timeslice* ad esso assegnato, avrebbe continuato ad usufruire del core sul quale era in esecuzione.
Il termine TPr (ΔT, i) media questo evento in rapporto al numero di voluntary context switch.

Sostituendo in EQ 7

$$p = \mathrm{TPr}(\Delta\mathrm{T}) \qquad \textbf{EQ 8}$$

per ogni thread, si ottiene:

$$\mathrm{S}(\Delta\mathrm{T}, \mathrm{N}, \mathrm{i}) = \frac{1}{(1 - \mathrm{TPr}(\Delta\mathrm{T}, \mathrm{i})) + \frac{\mathrm{TPr}(\Delta\mathrm{T}, \mathrm{i})}{N}} \qquad \textbf{EQ 9}$$

L'aumento di prestazione viene quindi allocato ad ogni singolo thread, in quantità proporzionale al suo numero di unvoluntary context switch.

L'uguaglianza di EQ 8 permette di definire il nuovo valore di unvoluntary context switch per ogni thread come segue:

$$U_{CSW}(\Delta\mathrm{T}, \mathrm{N}, \mathrm{i}) = \mathrm{S}(\Delta\mathrm{T}, \mathrm{N}, \mathrm{i}) * U_{CSW}(\Delta\mathrm{T}, \mathrm{i}) \qquad \textbf{EQ 10}$$

Sostituendo quindi EQ 10 in EQ 1 e EQ 6, si ottiene:

$$TPr(\Delta\mathrm{T}, \mathrm{N}) = \frac{U_{CSW}(N) * G_u}{U_{CSW}(\mathrm{N}) * G_u + V_{CSW} * G_v} \qquad \textbf{EQ 11}$$

$$\mathrm{SOr}(\Delta\mathrm{T}, \mathrm{N}) = \frac{\sum_{i=1}^{NThreads} TPr(\Delta\mathrm{T}, i, N)}{NThreads} \qquad \textbf{EQ 12}$$

Il nuovo valore di SOr così ottenuto rappresenta l'incremento di prestazioni atteso a livello di sistema, in funzione del numero di cores aggiunti.

Il valore di N deve tenere conto del numero di cores considerati come quantità di riferimento al momento del calcolo dell'aumento di prestazioni atteso.

In particolare, se i dati di partenza utilizzati per la predizione si riferiscono ad un sistema con un core, l'aggiunta di un ulteriore core significherà assegnare 2 alla variabile N, l'aggiunta di 2 core significherà assegnare 3 alla variabile N e così via.
Se, invece, i dati di partenza si riferiscono ad un sistema equipaggiato con 2 cores, l'aggiunta di un singolo core consisterà nell'assegnare a N il valore di 1.5.
In generale, si avrà:

$$\mathrm{N} = 1 + \frac{1}{N1} \qquad \textbf{EQ 13}$$

con N1 pari al valore iniziale di cores disponibili.
In Tabella 1 sono mostrati i valori di N nel caso di aggiunte di 2-16 cores in funzione del numero di cores iniziali.

Cores	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	1,00	2,00	3,00	4,00	5,00	6,00	7,00	8,00	9,00	10,00	11,00	12,00	13,00	14,00	15,00	16,00
2	0,50	1,00	1,50	2,00	2,50	3,00	3,50	4,00	4,50	5,00	5,50	6,00	6,50	7,00	7,50	8,00
3	0,33	0,67	1,00	1,33	1,67	2,00	2,33	2,67	3,00	3,33	3,67	4,00	4,33	4,67	5,00	5,33
4	0,25	0,50	0,75	1,00	1,25	1,50	1,75	2,00	2,25	2,50	2,75	3,00	3,25	3,50	3,75	4,00
5	0,20	0,40	0,60	0,80	1,00	1,20	1,40	1,60	1,80	2,00	2,20	2,40	2,60	2,80	3,00	3,20
6	0,17	0,33	0,50	0,67	0,83	1,00	1,17	1,33	1,50	1,67	1,83	2,00	2,17	2,33	2,50	2,67
7	0,14	0,29	0,43	0,57	0,71	0,86	1,00	1,14	1,29	1,43	1,57	1,71	1,86	2,00	2,14	2,29
8	0,13	0,25	0,38	0,50	0,63	0,75	0,88	1,00	1,13	1,25	1,38	1,50	1,63	1,75	1,88	2,00
9	0,11	0,22	0,33	0,44	0,56	0,67	0,78	0,89	1,00	1,11	1,22	1,33	1,44	1,56	1,67	1,78
10	0,10	0,20	0,30	0,40	0,50	0,60	0,70	0,80	0,90	1,00	1,10	1,20	1,30	1,40	1,50	1,60
11	0,09	0,18	0,27	0,36	0,45	0,55	0,64	0,73	0,82	0,91	1,00	1,09	1,18	1,27	1,36	1,45
12	0,08	0,17	0,25	0,33	0,42	0,50	0,58	0,67	0,75	0,83	0,92	1,00	1,08	1,17	1,25	1,33
13	0,08	0,15	0,23	0,31	0,38	0,46	0,54	0,62	0,69	0,77	0,85	0,92	1,00	1,08	1,15	1,23
14	0,07	0,14	0,21	0,29	0,36	0,43	0,50	0,57	0,64	0,71	0,79	0,86	0,93	1,00	1,07	1,14
15	0,07	0,13	0,20	0,27	0,33	0,40	0,47	0,53	0,60	0,67	0,73	0,80	0,87	0,93	1,00	1,07
16	0,06	0,13	0,19	0,25	0,31	0,38	0,44	0,50	0,56	0,63	0,69	0,75	0,81	0,88	0,94	1,00

Tabella 1 - Valore di N al variare del numero iniziale di cores disponibili

Capitolo 4

Validazione del Modello

4.1 Definizione degli use cases

La validazione del modello proposto si basa sull' acquisizione di dati di context switch misurati durante alcune sessioni di test utilizzando un sistema infotainment reale. Le funzionalità presenti in un sistema infotainment sono innumerevoli, pertanto, per collezionare dati che siano rappresentativi, è necessario definire accuratamente un insieme di casi d'uso (use cases) tipici per un utilizzatore del sistema.

I principali casi d'uso considerati sono i seguenti:

- Turn by turn navigation
- Route calculation
- Phone call
- Media player

- USB stick indexing
- Radio receiver

I casi d'uso non sono generalmente esclusivi, ma possono presentarsi in concorrenza: ad esempio, l'utente può seguire le indicazioni del navigatore mentre effettua una chiamata tramite le funzionalità *bluetooth*, o contemporaneamente all'*indexing* del contenuto di una chiavetta USB contenete molti files MP3, e così via.
Le sessioni di test sono state eseguite installando il sistema sotto analisi su un veicolo reale, in modo che le interfacce esterne, soprattutto relative alla rete veicolo, fossero attive.

Questo accorgimento, oltre a poter permettere ad un utilizzatore del sistema di comportarsi in modo "naturale" interagendo con il sistema, ha agevolato la generazione del corretto carico computazionale legato ai messaggi di rete provenienti dalle altre centraline presenti sul veicolo stesso e interagenti con il sistema di infotainment.

4.2 Ambiente di esecuzione e threads

L'architettura SW del sistema si basa su 176 threads eseguiti su Linux OS. Dal punto di vista HW, il sistema è basato su un System-on-Chip (SoC) embedded equipaggiato con 4 cores ARM e 1GB di RAM.

Per motivi di *storage durability*, nei sistemi infotainment automotive non è mai abilitata la funzionalità di swap su disco in caso di esaurimento della RAM.
La variabilità di prestazioni potenzialmente presente a causa di cicli di *memory swap* tra la RAM e il sottosistema di I/O è pertanto esclusa *by design*.

Le sessioni di test sono state eseguite abilitando un singolo core, e successivamente incrementando il numero di cores disponibili: è stato quindi possibile costruire quattro

"sistemi di riferimento", verso i quali verificare le predizioni del modello al variare del numero dei cores.

4.3 Analisi dei dati misurati

Durante le sessioni di test, della durata indicativa di un'ora l'una, il numero cumulativo di Unvoluntary Context Switch (U_{CSW}) e Voluntary Context Switch (V_{CSW}) per ogni thread presente sul sistema, è stato campionato ad una frequenza di 0,1Hz. È possibile quindi costruire una "impronta" statistica di attività per ognuno di essi. Ogni sessione di test è stata eseguita da un utente (driver) distinto e percorrendo percorsi distinti, in modo da arricchire il valore statistico dei dati raccolti.

Dal punto di vista globale, è anche possibile determinare una funzione di probabilità cumulativa (CDF) rappresentante la distribuzione del numero di U_{CSW} e V_{CSW} al termine di ogni sessione di test.

In Figure 4 e Figure 5 sono mostrati gli andamenti delle CDF dei Unvoluntary Context Switch e Voluntary Context Switch totali per ogni thread di due distinte sessioni di test, attivando un solo core.

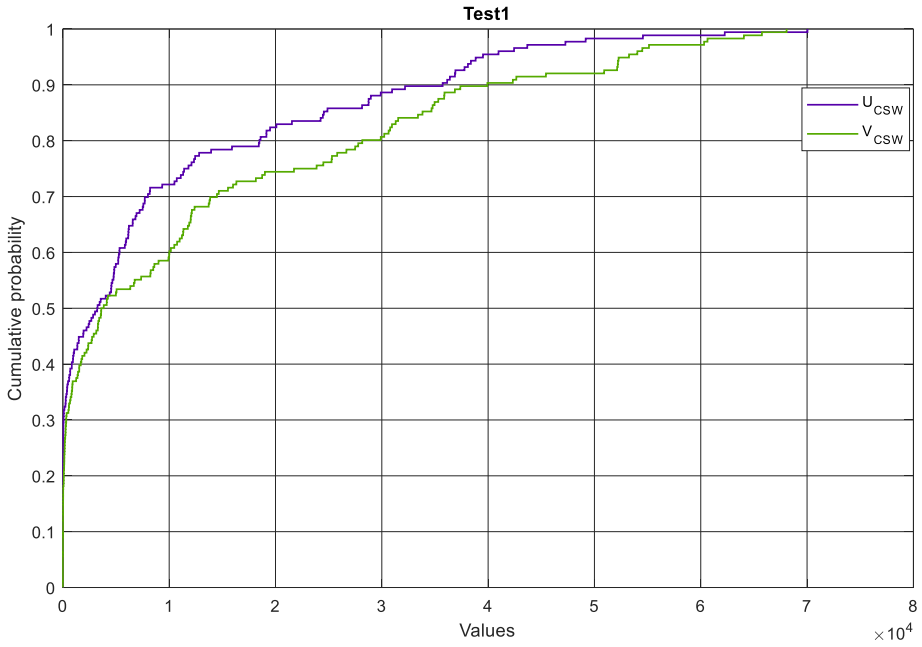

Figure 4 - CDF degli Unvoluntary e Voluntary CSW (1 core)

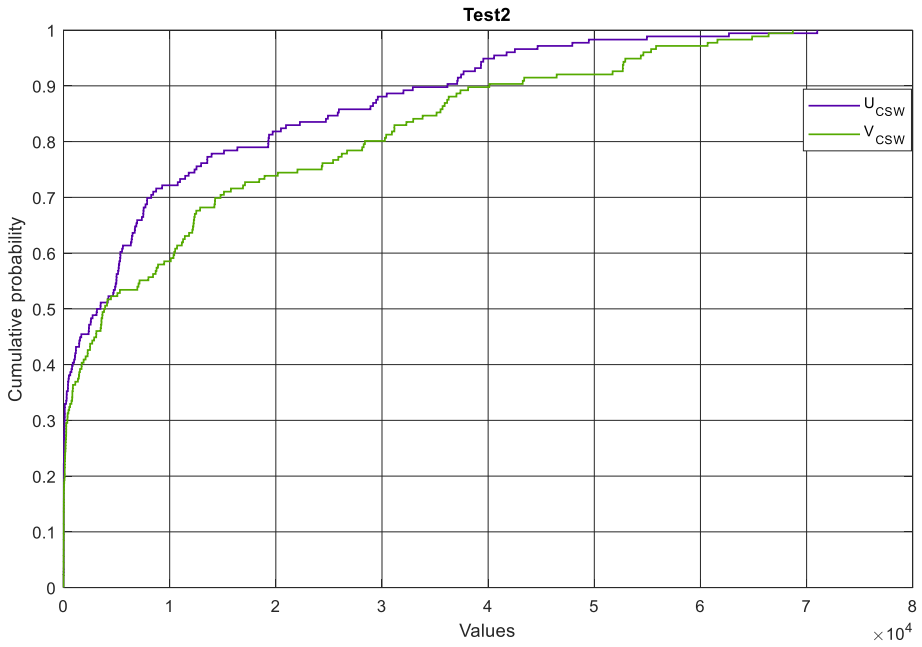

Figure 5 - CDF degli Unvoluntary e Voluntary CSW (1 core) - Seconda sessione

In Figure 6 sono mostrate le CDF di

$$Diff(U_{CSW}, \text{NTest}) = \text{ABS}(U_{CSW}(NTest = 1) - U_{CSW}(NTest = 2)) \quad \textbf{EQ 14}$$

$$Diff(V_{CSW}, \text{NTest}) = \text{ABS}(V_{CSW}(NTest = 1) - V_{CSW}(NTest = 2)) \quad \textbf{EQ 15}$$

Si può notare che le due sessioni di test sono confrontabili in termini di valori di U_{CSW} e V_{CSW}: questa circostanza avvalora il significato statistico dei test condotti, evidenziando che l'osservazione del comportamento del sistema è stato effettuato a valle degli stati transitori iniziali.

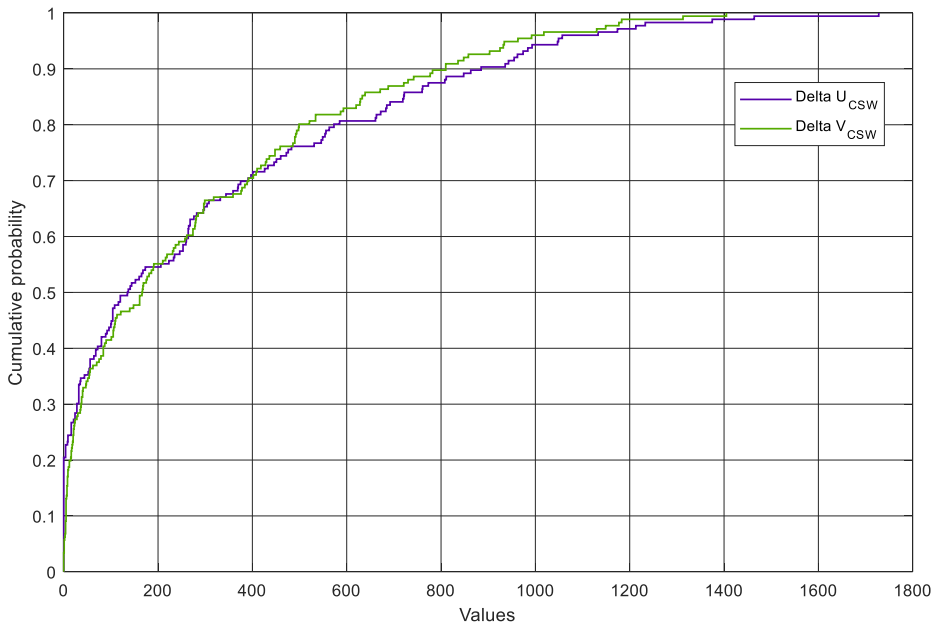

Figure 6 – Confronto tra Unvoluntary e Voluntary Context Switch tra due sessioni di test distinte (1 core)

4.4 Da un core a quattro cores

Analizzando i dati raccolti utilizzando tutti e quattro i cores disponibili nel SoC, ci aspettiamo di vedere una crescita del numero di Unvoluntary Context Switch dovuti al maggior numero di thread eseguiti contemporaneamente, e parallelamente una crescita anche del numero di Voluntary Context Switch: sia per il maggior numero di thread eseguiti in unità di tempo, sia per l'aumento delle risoluzioni di attese su semafori, locks, code, etc…

L'andamento dei dati reali è mostrato in Figure 7 e Figure 8. Esso coincide con il comportamento atteso. La CDF dei dati relativi all'utilizzo di 4 cores risulta, infatti, sotto la CDF per 1 singolo core, ad indicare la maggiore probabilità di incontrare valori di U_{CSW} e V_{CSW} superiori.
È stata usata la rappresentazione logaritmica sull'asse X per meglio apprezzare il fattore di scala tra le due misurazioni.

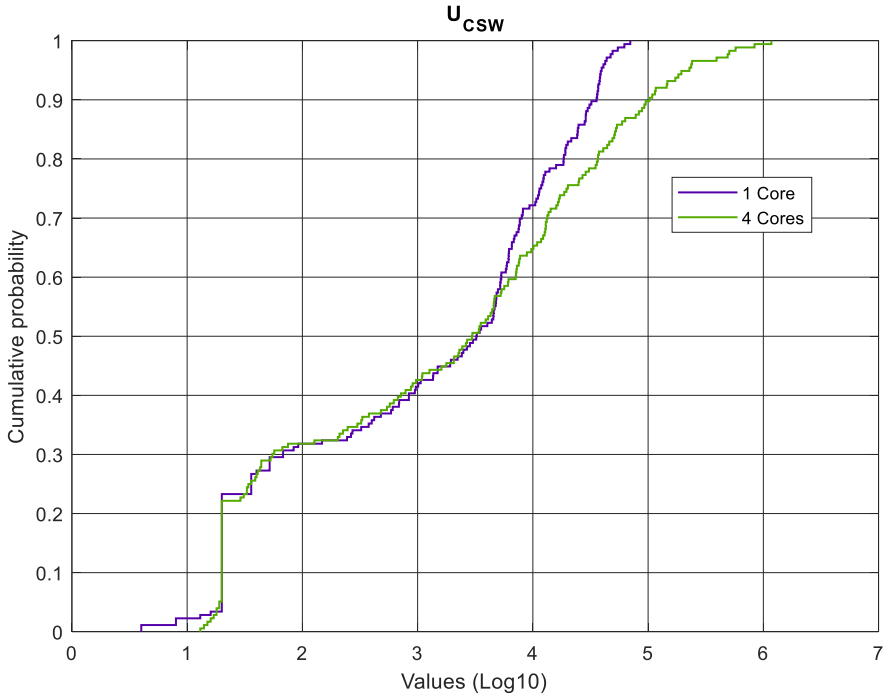

Figure 7 - CDF degli Unvoluntary Context Switch per 1 e 4 cores

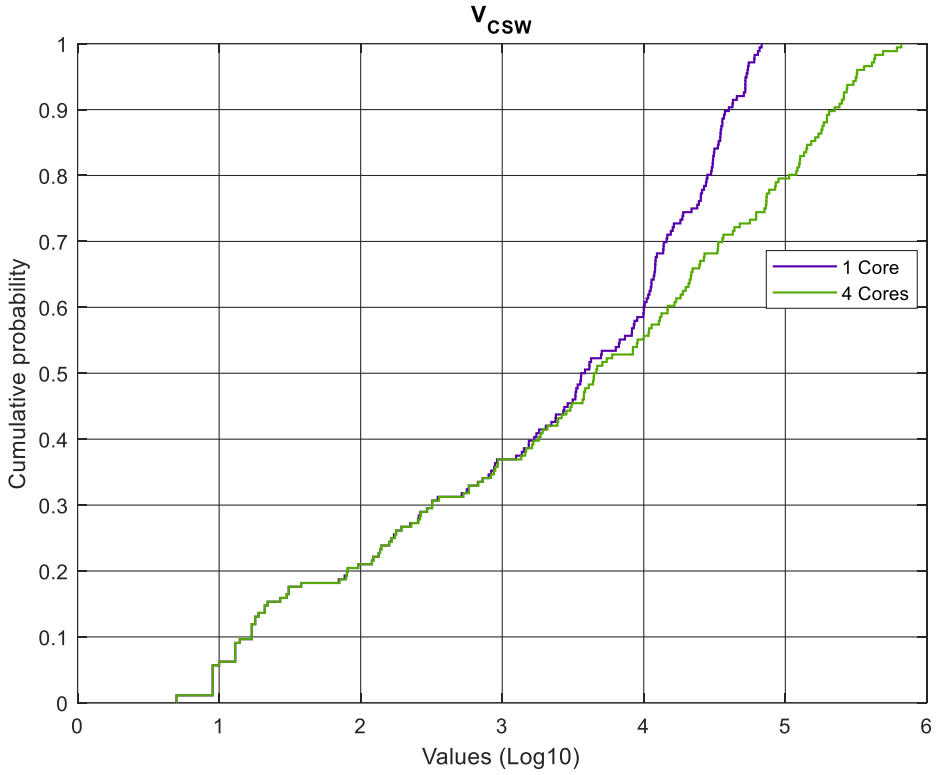

Figure 8 - CDF dei Voluntary Context Switch per 1 e 4 cores

Il vero indicatore da valutare, in questo caso, è la variazione del valore di TPr, introdotto nel Capitolo 3.

$$TPr(\Delta T) = \frac{U_{CSW} * G_u}{U_{CSW} * G_u + V_{CSW} * G_v} \qquad \textbf{EQ 16}$$

Il comportamento atteso del parametro, dal punto di vista statistico, è la presenza di una CDP di $TPr(\Delta T)_{|4\ cores}$ sostanzialmente sopra la CDP di $TPr(\Delta T)_{|1\ core}$, a significare che il sistema, nel suo complesso tende a diventare "relativamente" più I/O-Bound – i processi tendono ad avere un TPr più basso – a causa del ridursi del numero di Unvoluntary Context Switch grazie alla disponibilità di più cores, in accordo con la formula

La Figure 9 mostra che i dati sperimentali seguono l'aspettativa del modello.

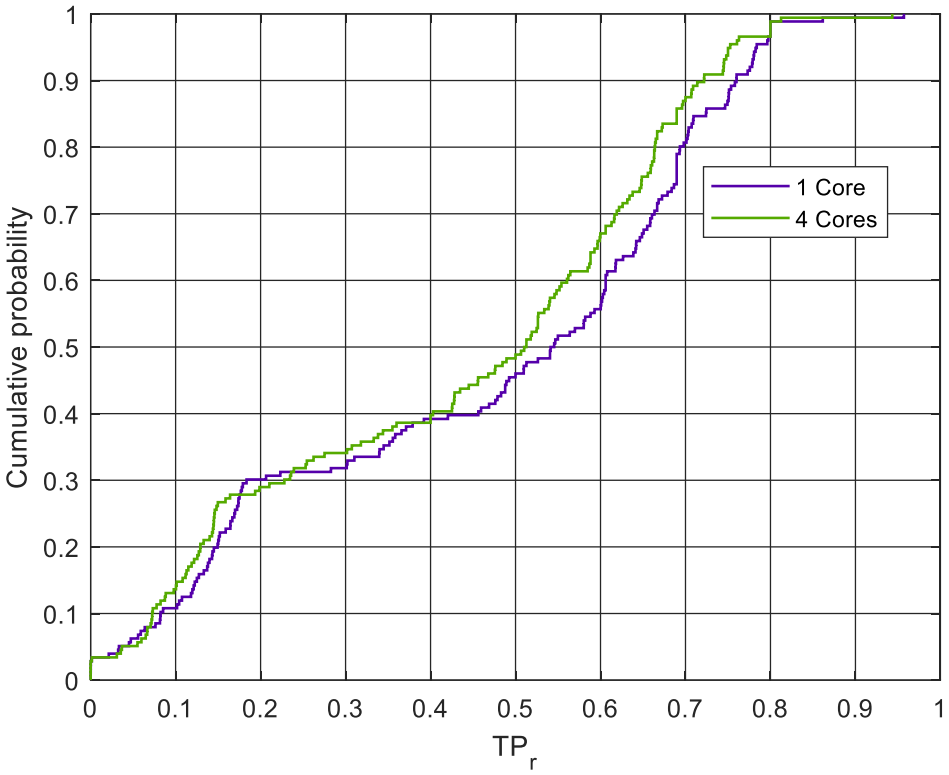

Figure 9 - Variazione del parametro TPr al variare del numero di cores

4.5 Applicazione del modello ai dati sperimentali

Il modello proposto prevede di utilizzare il valore di TPr di ogni processo in esecuzione come valore di *p* nella formula di Amdahl.
Ciò significa che la nuova interpretazione data al valore di *p* si declina dalla percentuale del processo che può essere parallelizzato alla percentuale di *CPU-Boundariness* che verrebbe soddisfatta dalla presenza di un ulteriore core a disposizione.

Dal punto di vista teorico, questa ipotesi è suffragata dal fatto che il numero di Unvoluntary Context Switch di un processo, come visto nei paragrafi precedenti, tende

a diminuire all'aumentare del numero di cores disponibili, andando ad erodere, man mano che il numero di cores cresce, il potenziale vantaggio dell'introduzione di una risorsa di esecuzione aggiuntiva.

Man mano che il numero di Unvoluntary Context Switch tende a ridursi, anche il valore di TPr si riduce: l'effetto ottenuto segue lo stesso comportamento atteso dalla progressiva diminuzione della parallelizzazione di un processo, fino ad arrivare al punto di non poter più parallelizzare alcuna parte di codice.

L'applicazione del modello prevede quindi di ricalcolare i valori attesi di TPr per ogni processo applicando la legge di Amdahl, e con i nuovi valori così ottenuti, calcolare il valore di System Overbooking Ratio nel nuovo scenario.

Avremmo quindi un valore $\mathrm{SOr}(\Delta T, N)_{|\mathrm{Modello}}$ da confrontare con il valore calcolato utilizzando i dati reali, al variare di N.

I dati reali, misurati utilizzando un singolo core, saranno quindi inseriti nel modello per prevedere l'andamento di SOr (ΔT, N) con N=2,3,4, calcolando quindi l'errore di predizione $\varepsilon_{1,N}$:

$$\forall N = \{2,3,4\} \qquad \varepsilon_{1,N} = |\mathrm{SOr}(\Delta T, N) - \mathrm{SOr}(\Delta T, N)_{Modello}| \qquad \textbf{EQ 17}$$

In generale, per quanto riguarda i dati reali acquisiti durante i test nei quali sono stati attivati M cores, essi saranno utilizzati allo stesso modo per prevedere l'andamento di SOr (ΔT, N) con N=M+1, … ,4 calcolando anche per essi l'errore di predizione $\varepsilon_{M,N}$

$$\forall N = \{M+1, \dots, 4\} \qquad \varepsilon_{M,N} = |\mathrm{SOr}(\Delta T, N) - \mathrm{SOr}(\Delta T, N)_{Modello}| \qquad \textbf{EQ 18}$$

Nella Tabella 2 sono mostrati i valori di $\mathrm{SOr}(\Delta T, N)_{Modello}$, $\mathrm{SOr}(\Delta T, N)$ e $\varepsilon_{M,N}$ per M=1 e N= {1,2,3,4};

N	$SOr(\Delta T, N)_{Modello}$	$SOr(\Delta T, N)$	$\varepsilon_{M,N}$	$\%\varepsilon_{M,N}$
1	0.4190	0.4173	0.0017	0.4112
2	0.4260	0.4227	0.0033	0.7754
3	0.4380	0.4368	0.0012	0.2771
4	0.4556	0.4561	0.0005	0.1169

Tabella 2 - SOr stimato e calcolato dai dati reali per M=1 e N=1,2,3,4

Come si può dedurre dalla Tabella 2, il valore di SOr predetto dal modello per i cores 1,2,3,4 proiettando i dati provenienti dal test con un singolo core sono molto simili a quelli effettivamente misurati sul sistema reale.

In Figure 10 e Figure 11 sono mostrati i dati in forma grafica.

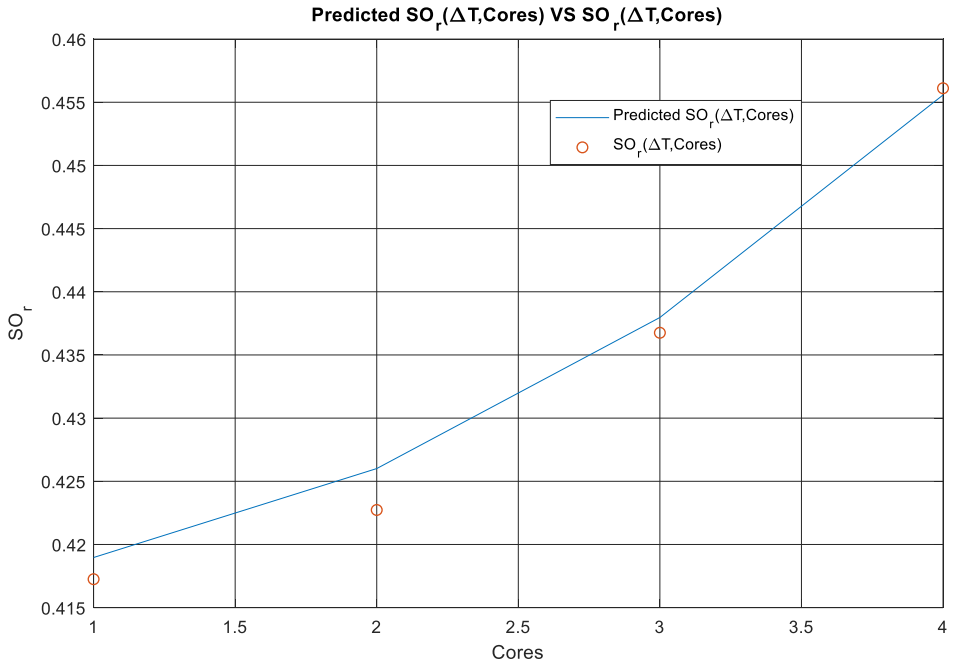

Figure 10 - Valori di SOr calcolati dai dati reali e predetti dal modello a partire dai dati riferiti ad un singolo core

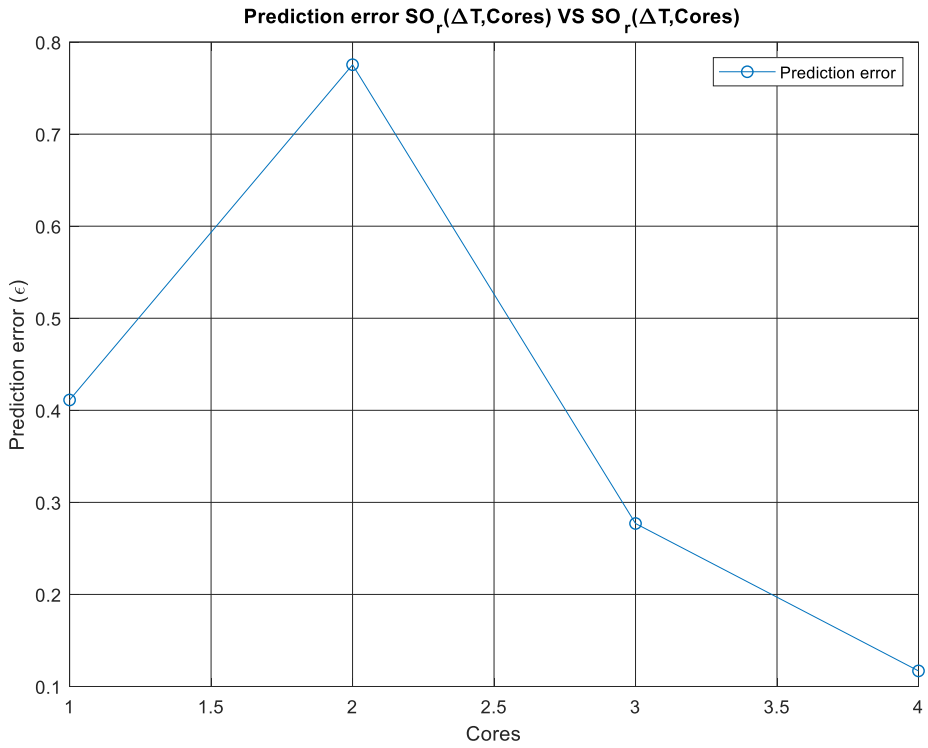

Figure 11 - Percentuale di errore di predizione del modello

In Tabella 3 sono mostrati, in forma tabellare, i risultati del modello applicato ai dati della seconda sessione di test, anche in questo caso, inserendo nel modello i dati relativi all'esecuzione dei thread con un singolo core a disposizione.

N	$SOr(\Delta T, N)_{Modello}$	$SOr(\Delta T, N)$	$\varepsilon_{M,N}$	$\%\varepsilon_{M,N}$
1	0.4179	0.4176	0.0003	0.0892
2	0.4226	0.4222	0.0004	0.0967
3	0.4385	0.4356	0.0028	0.6530
4	0.4584	0.4543	0.0041	0.9098

Tabella 3 - SOr stimato e calcolato dai dati reali per M=1 e N=1,2,3,4 (Test2)

In Figure 12 e Figure 13 sono mostrati i dati per il Test2 in forma grafica.

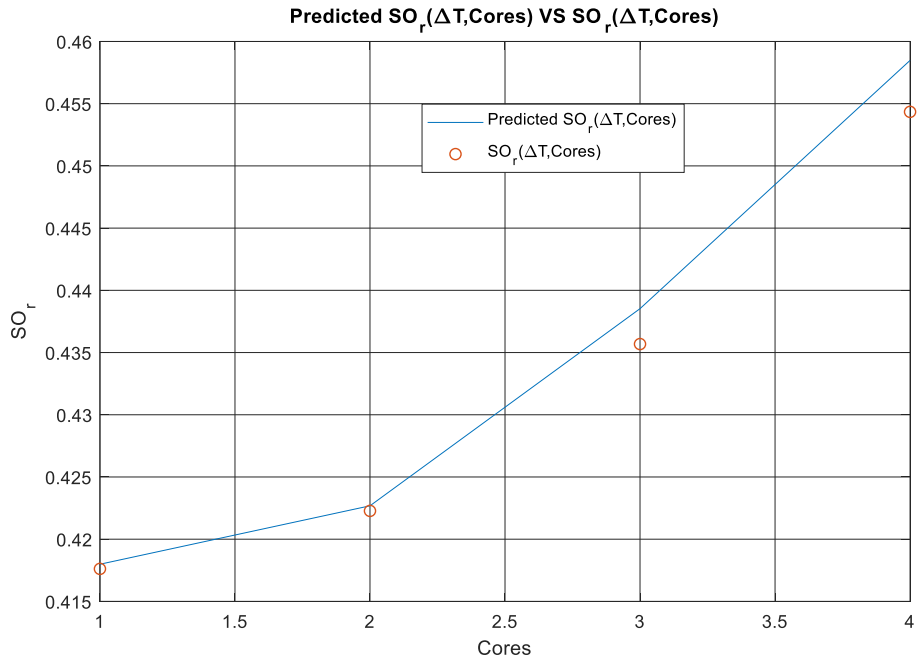

Figure 12 - Valori di SOr calcolati dai dati reali e predetti dal modello a partire dai dati riferiti ad un singolo core (Test2)

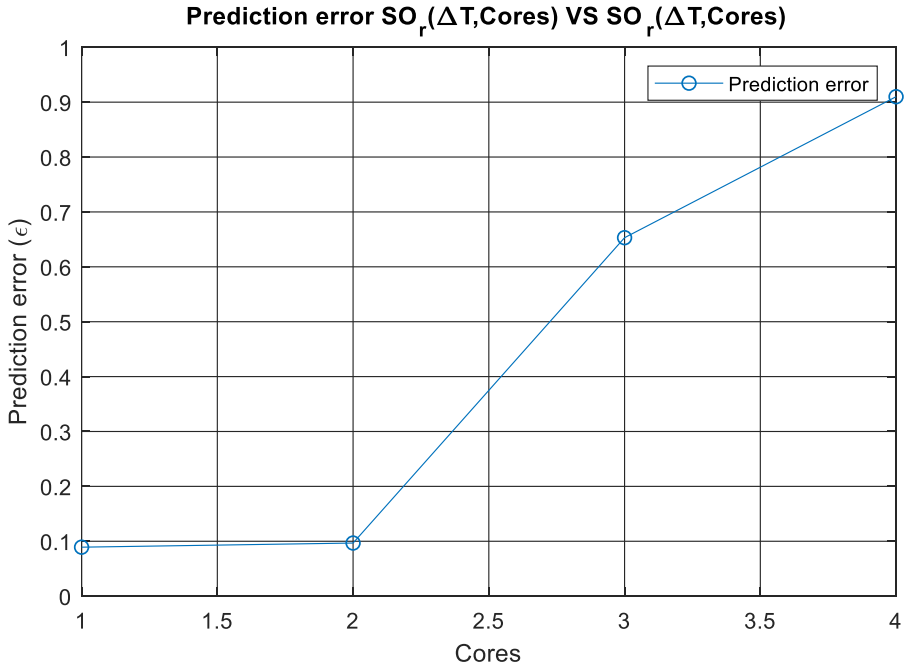

Figure 13 - Percentuale di errore di predizione del modello (Test2)

Come precedentemente discusso, i processi di previsione è stato applicato utilizzando anche i dati raccolti nei test dove erano attivi un numero variabile di cores, fino a 4. Nella Figure 14 - Valori di SOr calcolati dai dati reali e predetti dal modello sono mostrati i risultati ottenuti.

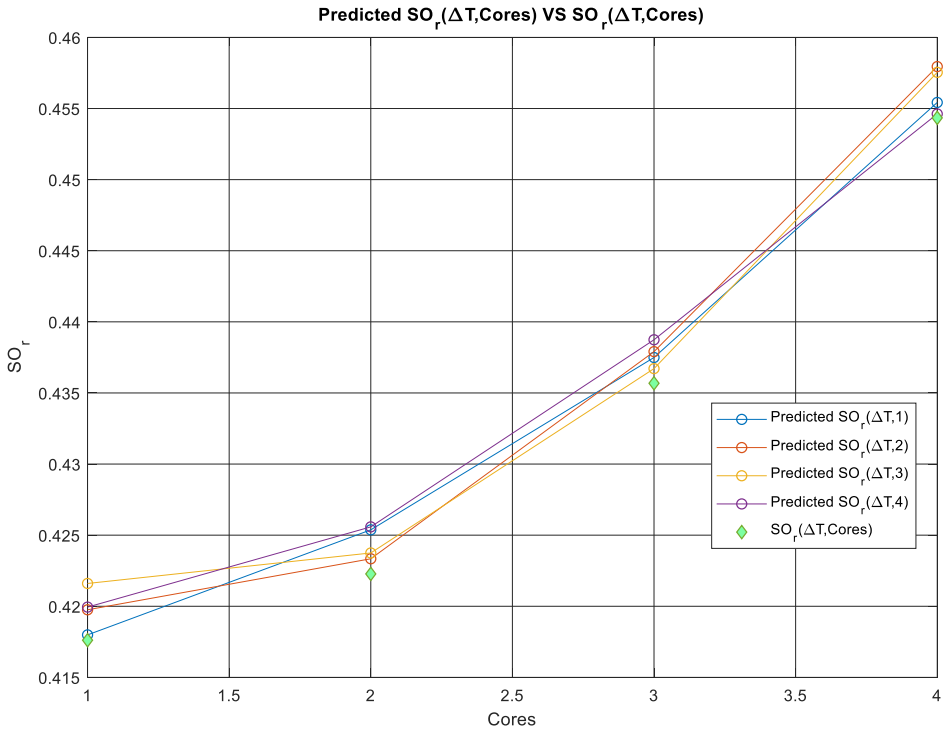

Figure 14 - Valori di SOr calcolati dai dati reali e predetti dal modello

I risultati ottenuti nella stima dei valori di SOr(ΔT) in situazioni sia single-core, sia multi-core, suggerisce la possibilità di utilizzare i parametri studiati come previsione attendibile sull'eventuale beneficio, a livello di sistema, a fronte dell'introduzione o meno di un ulteriore core (e il relativo costo aggiuntivo).

È interessante notare che il modello è in grado di prevedere anche, almeno a livello qualitativo e di confronto, quale potrebbe essere l'impatto negativo di prestazioni in caso di riduzione costi, e conseguentemente, di riduzione dei cores disponibili.

Infatti il grafico di Figure 14 mostra che è possibile stimare anche "a ritroso" l'eventuale downgrade di performance, ad esempio misurando i dati reali su un sistema

a 4 cores e proiettando l'andamento del parametro SOr(ΔT) su un numero di cores inferiore.

4.6 Predizioni oltre i 4 cores

Il modello predittivo è stato fino a qui utilizzato per confrontare le previsioni con i dati reali raccolti sul campo.
Il confronto con i dati reali e le previsioni del modello evidenzia una buona capacità predittiva, sia nel caso l'attenzione sia focalizzata a capire se sia ragionevole sostenere dei costi aggiuntivi per poter disporre di un sistema hardware più potente per ottenere migliori prestazioni del software, sia per predire eventuali impatti negativi sulle performance del sistema in caso di riduzione costi e, conseguentemente di uso di hardware meno performante.

Ora l'attenzione si sposta sulla capacità del modello di predire una soglia massima di aumento di prestazioni del sistema, oltre la quale il guadagno in termini di performance non giustificherebbe il delta costo da sostenere.

Allo scopo, è stato analizzato l'andamento del coefficiente SOr(ΔT) applicando il modello fino a 16 cores, in modo da analizzare l'andamento generale evidenziando eventuali valori massimi e/o minimi della curva associata.

Nella Figure 15 è mostrata la curva $\mathrm{SOr}(\Delta T, N)_{Modello}$ applicando i modello sui dati estratti dal Test1 attivando un solo core.
L'analisi della curva rivela un picco di prestazioni aggiuntive intorno ai 6 cores, oltre il quale, il guadagno ottenibile incrementando il numero di cores tende a diminuire rapidamente fino a raggiungere un asintoto a zero.

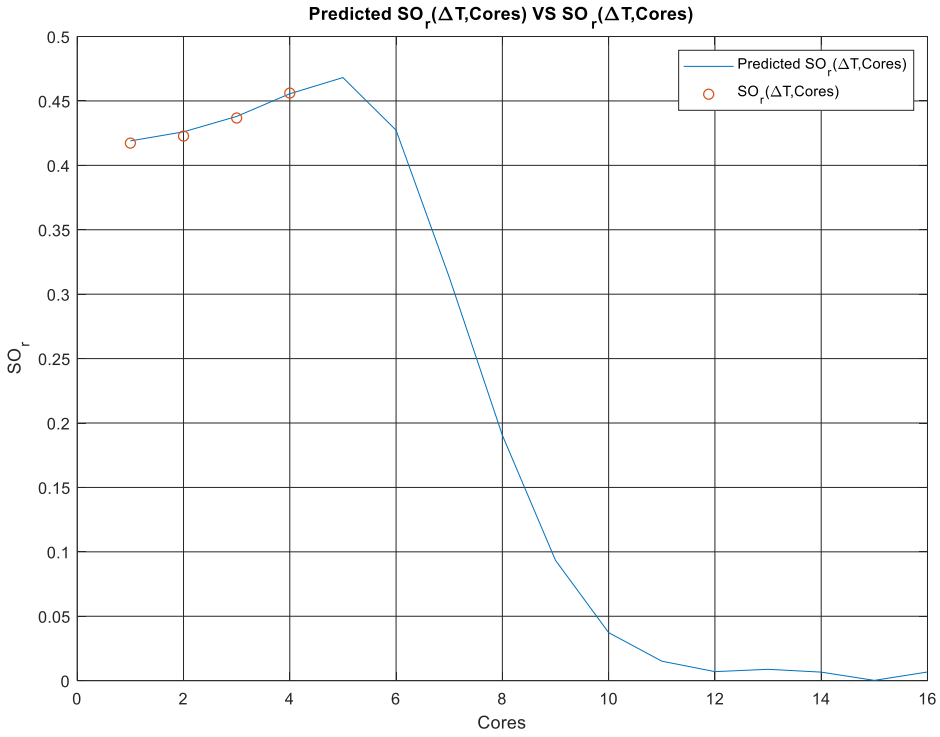

Figure 15 - Andamento della predizione di SOr(ΔT) al variare del numero di cores

Nella Figure 16 sono mostrati gli andamenti delle curve per M = {1,2,3,4}, cioè applicando il modello di volta in volta sui dati raccolti con il Test1 variando il numero di cores effettivamente attivi.

Si può notare che l'andamento delle 4 curve è sovrapponibile: il modello risulta stabile nella previsione dell'andamento di $\mathrm{SOr}(\Delta T, N)_{Modello}$ al variare del numero di cores reali con i quali sono stati raccolti i dati di partenza.

Questa congruenza di curve permette di avvalorare il possibile uso del modello indipendentemente dal numero di cores con i quali si raccolgono i dati da sottoporre ad analisi. Ciò significa che non vi è un requisito su quale debba essere la configurazione hardware iniziale per la raccolta dei dati.

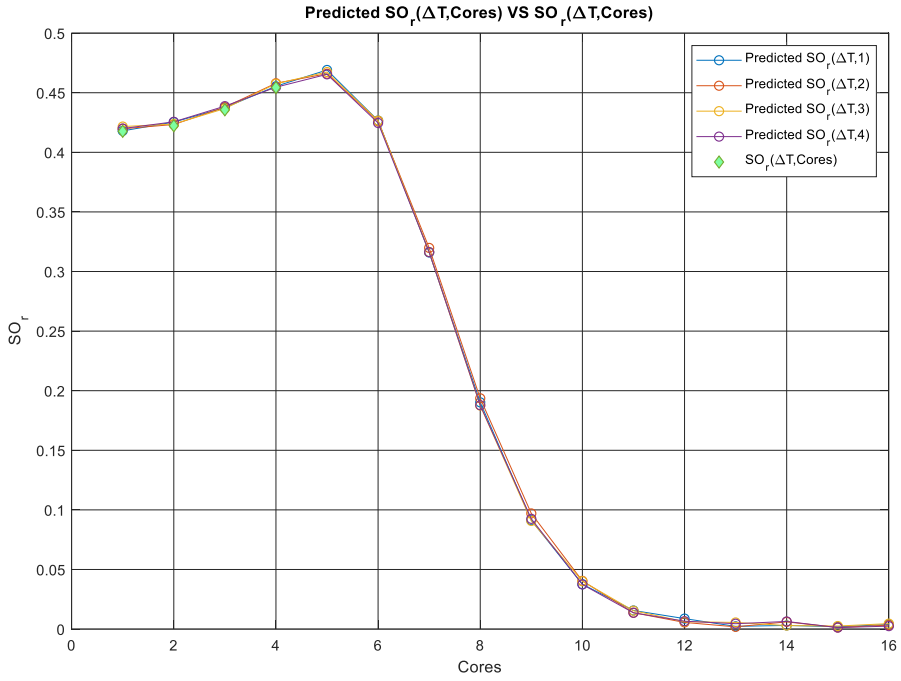

Figure 16 - Andamento della predizione di SOr(ΔT) al variare del numero di cores

Nella Figure 17 è mostrato un dettaglio della sezione con N = {5-12}.

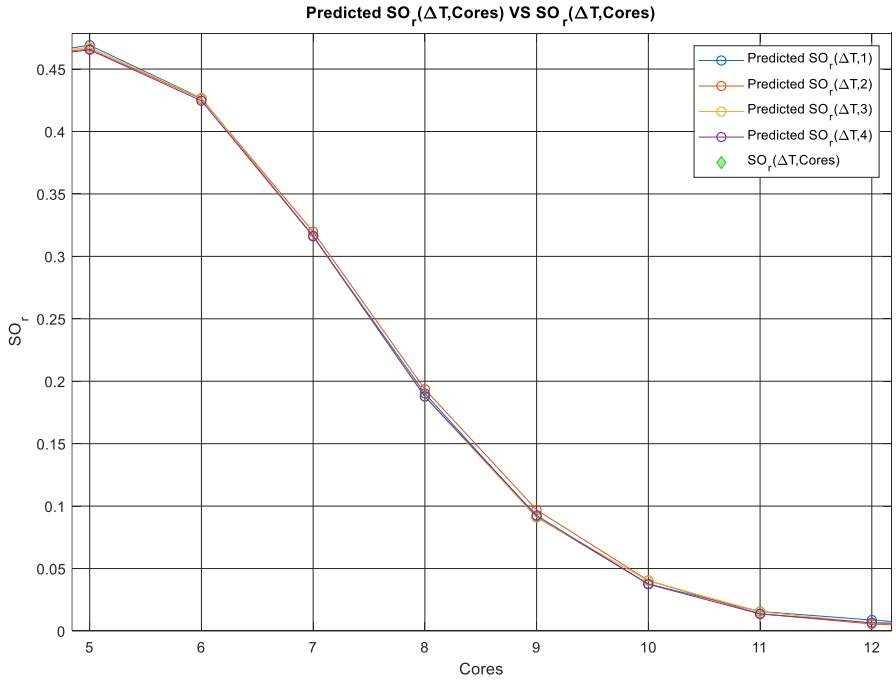

Figure 17 -Andamento della predizione di SOr(ΔT) al variare del numero di cores (Dettaglio)

4.7 Interpolazione del modello

Spesso è utile poter utilizzare un'espressone analitica del modello che permetta di esprimere l'andamento della curva di previsione in forma chiusa.

L'espressione analitica fotografa uno stato del software derivante dall'analisi del suo comportamento, pertanto la validità del risultato di curve fitting varrà nello specifico scenario studiato: in caso di forti cambiamenti alla struttura del software, sarà necessario eseguire almeno una sessione di test significativa per raccogliere i nuovi dati da imporre al modello per una corretta predizione.

A valle dell'analisi, è comunque interessante poter fruire di una singola espressione matematica che sintetizzi il risultato dell'analisi.

Analizzando l'andamento della curva, e postulando un valore di:

$$\begin{cases} SOr(\Delta T) = 0 \; per \; NCores = 0 \\ SOr(\Delta T) \; \to 0 \; per \; NCores \; \to \infty \end{cases} \qquad \textbf{EQ 19}$$

è possibile considerare un modello razionale della forma:

$$Y = \frac{p_1 x^n + \cdots + p_{n+1}}{x^n + q_1 x^{n-1} + q_{n-1}} \qquad \textbf{EQ 20}$$

Matlab permette di calcolare alcuni parametri per verificare l'aderenza del fitting con i dati di modello.

In particolare sono stati presi in considerazione i seguenti indicatori:

- Sum of Square Due to Error (SSE)
- R-Square
- Root Mean Squared Error (RMSE)

Sum of Square Due To Error (SSE) è definita come segue:

$$SSE = \sum_{i=1}^{n} w_i (y_i - \hat{y}_i) \qquad \textbf{EQ 21}$$

Esso rappresenta la deviazione totale dei valori espressi dalla curva di fitting rispetto ai valori originali. Più il valore di SSE tende a zero, più la curva di fitting rappresenta bene i dati di ingresso.

R-Square tiene conto di quanto il fitting ottenuto segue la variazione dei dati in ingresso. Esso è definito come segue:

$$SST = \sum_{i=1}^{n} w_i (y_i - \hat{y}_i)^2 \qquad \textbf{EQ 22}$$

$$R \; Square = 1 - \frac{SSE}{SST} \qquad \textbf{EQ 23}$$

Il valore di R-Square può variare nell'intervallo [0,1]. Più il valore si avvicina a 1, più il modello di fitting tiene conto di una percentuale maggiore di varianza dei dati in ingresso.

Il Root Mean Squared Error può essere definita come la stima della deviazione standard della componente casuale nei dati di ingresso.
Essa è definita come segue:

$$MSE = \frac{SSE}{v} \qquad \textbf{EQ 24}$$

$$MRSE = \sqrt{MSE} \qquad \textbf{EQ 25}$$

Anche in questo caso, più il Root Mean Squared Error tende a zero, più la curva di fitting rappresenta bene i dati di ingresso.

L'insieme dei parametri elencati rappresenta la così detta *goodness of fit*, cioè la bontà del modello analitico ottenuto.
Applicando il fitting tool disponibile su Matlab, riducendo al minimo il numero di parametri compatibilmente con una goodness of fit sufficiente, si ottengono i seguenti parametri:

Parametro	Valore
p1	0,06417
p2	-1,612
p3	8,86
p4	16,74
p5	8,307
p6	-6,179
q1	-12,81
q2	48,71
q3	14,99
q4	4,447
q5	6,321

Il polinomio razionale risultante è composto quindi da 2 polinomi, uno al numeratore, l'altro al denominatore, di grado 5.

I valori del fitting che sono stati ottenuti sono i seguenti:

- SSE: $5.911*10^{-5}$
- R-Square: 0.9999
- RMSE: 0.003438

Sia SSE che RMSE sono molto vicini a zero, mentre R-Square è molto vicino a 1: la goodness of fit generale può essere considerata molto affidabile.
In Figure 18 è mostrato il risultato del modello di fitting: come si può notare, i vincoli espressi nelle equazioni EQ 19 sono rispettate.

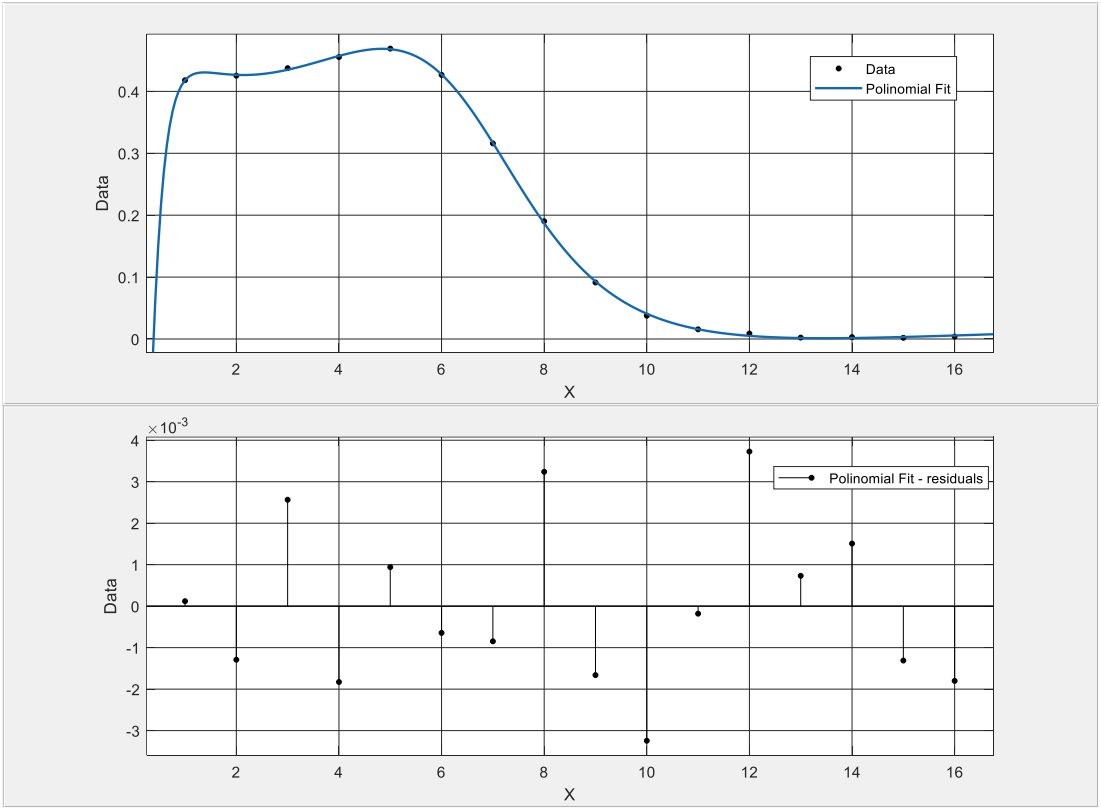

Figure 18 - Risultato del fitting dei dati del modello

Capitolo 5

Simulatore di sistema

5.1 Introduzione

Lo studio del modello di predizione ha permesso di verificare se, attraverso la profilatura del numero di unvoluntary e voluntary context switch di ogni thread, fosse possibile costruire un sistema di predizione del vantaggio (o del degrado) prestazionale del sistema a fronte di un incremento (o del decremento) del numero di cores, potendo quindi valutare se il rapporto tra il costo aggiuntivo da sostenere per introdurre la modifica e l'incremento prestazionale, in un caso, o il rapporto tra il risparmio ottenuto e il degrado prestazionale, nell'altro, sia vantaggioso.

La raccolta dei dati è però generalmente dispendiosa, sia in termini di tempo, sia in costi di esercizio poiché ogni sessione richiede almeno:

- l'esecuzione del software su un sistema installato su un veicolo reale;

- alcune modifiche del kernel per la raccolta dei dati;
- il coinvolgimento di drivers e utenti che utilizzino fisicamente il sistema;
- la verifica che i più importanti use cases di interesse siano effettivamente eseguiti.

Vi sono poi alcuni altri aspetti da tenere in considerazione:

- in fase di sviluppo, i test dovrebbero essere eseguiti ad ogni release in modo da monitorare eventuali modifiche nel comportamento del sistema
- non è sempre facile garantire la ripetibilità dei test e la possibilità di un confronto dei risultati a lungo termine
- il design degli *use cases* può dover essere ridefinito in caso di modifica dei requisiti cliente, obbligando di fatto a rieseguire i test considerando anche i nuovi scenari

Da queste considerazioni si evince che un sistema di simulazione che permetta di generare i dati necessari per l'utilizzo del modello possa rivelarsi molto utile.
In particolare, permetterebbe di simulare alcuni use cases limite e/o sperimentare a priori l'impatto sulle prestazioni del sistema dell'introduzione di una nuova funzionalità, conoscendone i profili di "CPU-Bound" e "I/O-Bound".

Per poter raggiungere questo obiettivo, è necessario poter disporre dei seguenti elementi:

- un profilo statistico di "CPU-Bound" e "I/O-Bound" per ogni thread o gruppo di thread coinvolti in una specifica funzionalità;
- un sistema di simulazione in grado, a partire dai profili di ogni thread, di generare i valori di Unvoluntary Context Switch a Voluntary Context Switch associati, attraverso esecuzioni "virtuali" dei theads.

Il sistema di simulazione deve poter essere configurabile in modo da considerare un numero variabile di cores.

5.2 Profilo statistico dei threads

Il modello di riferimento è basato, come descritto nei capitoli precedenti, sui parametri di Unvoluntary Context Switch (U_{CSW}) e Voluntary Context Switch (V_{CSW}) dei thread. Come mostrato nei Capitoli 3 e 4, il parametro TPr(ΔT) è direttamente legato al rapporto tra i due tipi di context switch, pertanto è utile poter conoscere la distribuzione generale e la tipologia dei thread che compongono il sistema nel suo complesso.

Analizzando i dati misurati sul sistema reale, in una finestra temporale ΔT, utilizzando un singolo core per l'esecuzione dei thread, si può notare che, a parte alcuni thread molto "I/O-Bound", che si posizionano in sulla sinistra del grafico di Figure 19, un buon numero di thread presenta un rapporto tra Unvoluntary Context Switch e Voluntary Context Switch tendente a 1 (Figure 19)

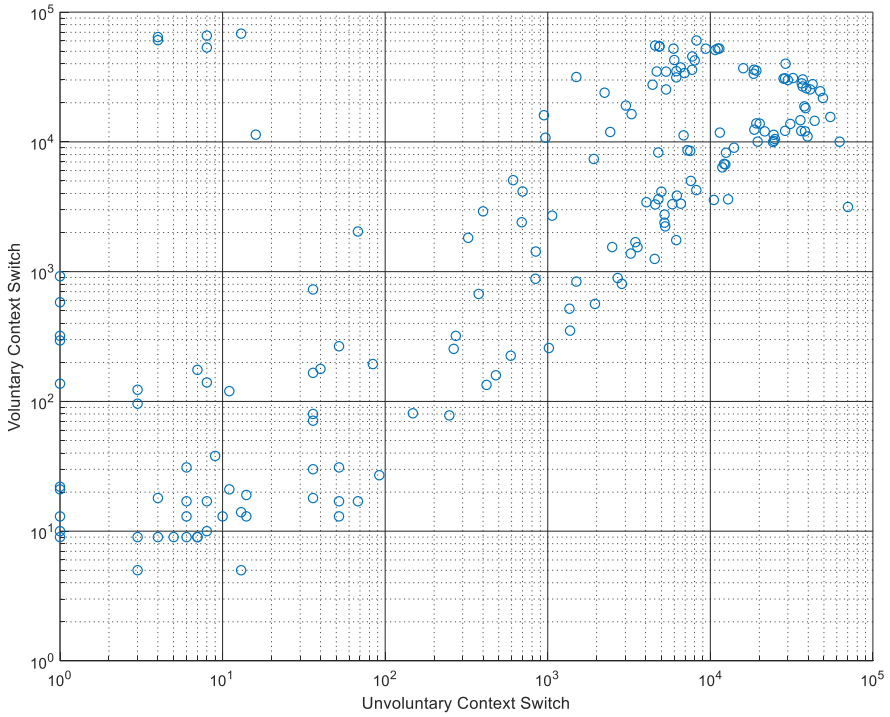

Figure 19 - Scatter plot dei thread in funzione dei U_{CSW} e V_{CSW}. Gli assi sono in scala logaritmica.

Dal grafico di Figure 19 si evince anche che vi sono un certo numero di thread particolarmente attivi (in alto a destra), presentando un notevole impatto sulle risorse di CPU, avendo essi un altissimo valore di U_{CSW}, ma al contempo, anche un elevato impatto sui sottosistemi di I/O.
Al contrario, non sono evidenti thread "CPU-Bound" puri, i quali si presenterebbero in basso a destra, sull'asse delle ascisse.

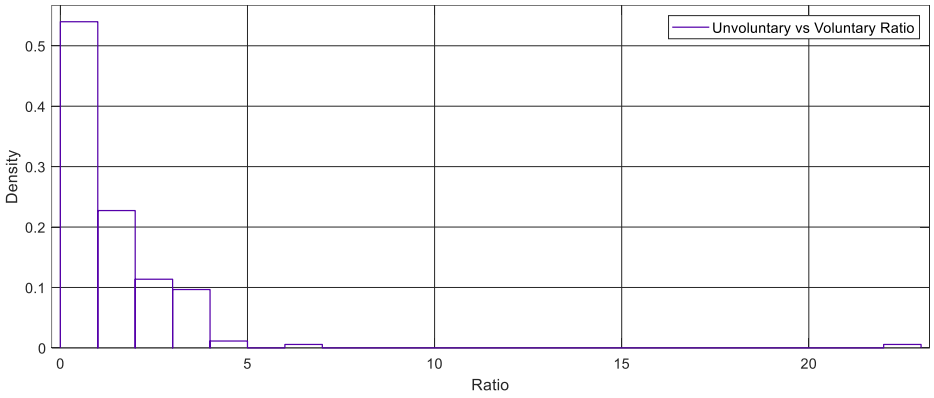

Figure 20 - Rapporto tra Unvoluntary e Voluntary Context Switch

La stessa analisi può essere svolta considerando i dati raccolti abilitando 4 cores. In Figure 21 si nota sia un aumento del numero di Unvoluntary Context Switch dovuto al maggior numero di thread eseguiti per unità di tempo, sia un aumento di concentrazione di thread che presentano un valore di V_{CSW} superiore al valore di U_{CSW}. In particolare, questi thread iniziano a percepire l'effetto della presenza dei cores aggiuntivi in maniera più significativa, riducendo di fatto il loro valore intrinseco di TPr.

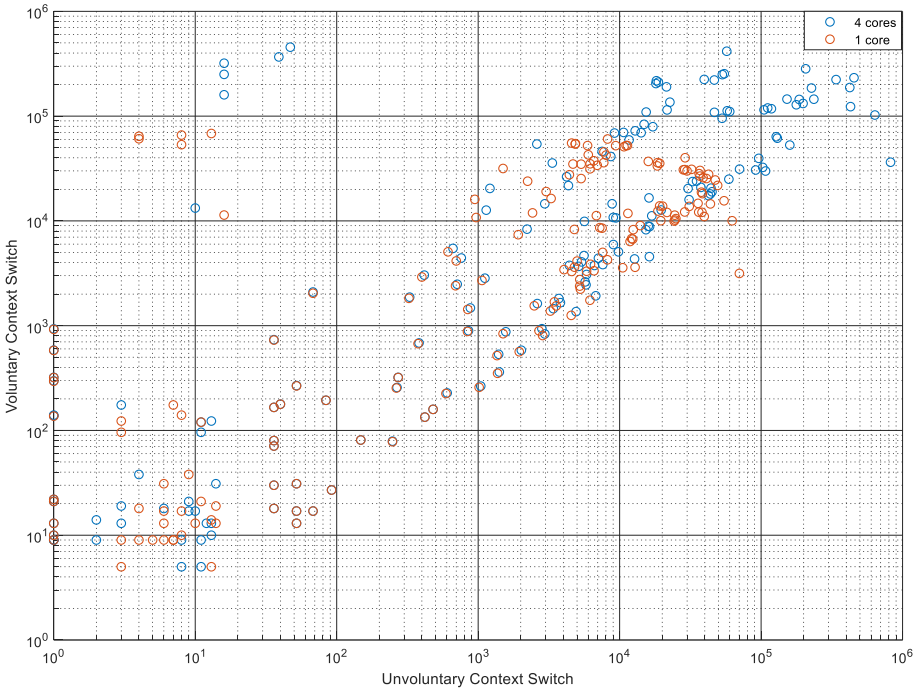

Figure 21 - Scatter plot dei thread in funzione dei U_{CSW} e V_{CSW} per 1 e 4 cores. Gli assi sono in scala logaritmica.

Dalla Figure 22 si può evincere che un buon numero di thread presenta un rapporto tra Unvoluntary Context Switch e Voluntary Context Switch tendente a 0.3-0.4.

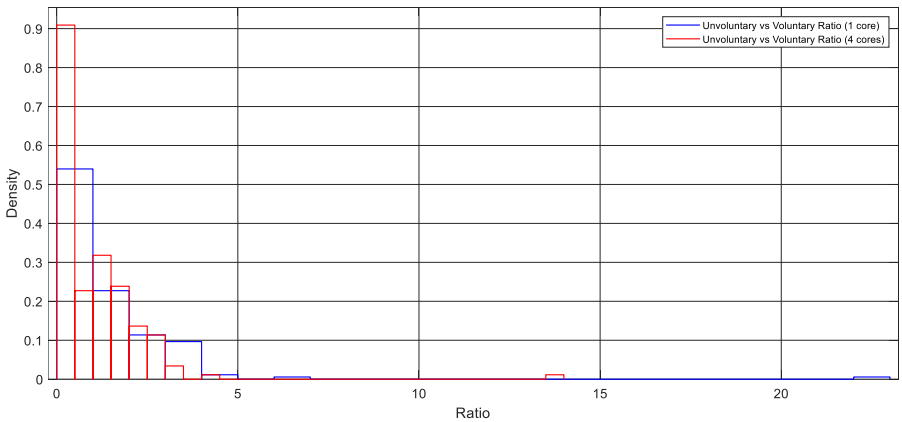

Figure 22 - Rapporto tra Unvoluntary e Voluntary Context Switch (1-4 cores)

Per poter avere un quadro generale delle caratteristiche dei thread in esecuzione, è necessario anche capire le sequenze temporali degli U_{CSW} e V_{CSW}, sia per quanto riguarda l'alternarsi dei due tipi di context switch, sia relativo alla distanza temporale tra due context switch dello stesso tipo.
Questi valori permettono di capire il profilo di effettiva attività del singolo thread rispetto ai momenti di *idle*, dove il thread, a fronte di una chiamata di sistema, è in attesa dell'esito della stessa.

In uno scenario multi-core, oltretutto, se un thread esaurisce il suo tempo di esecuzione ma, a fronte di un numero limitato di thread nella coda *ready*, viene successivamente ripescato dall'algoritmo di scheduling per una continuazione dell'esecuzione, avrà la possibilità di distanziare i due eventi temporali che lo vedono protagonista di un unvoluntary context switch.
Questo parametro permette di capire le relazioni temporali di esecuzione tra i thread e, indirettamente, la concorrenza reale tra essi per accedere alle CPU disponibili.

Un primo sguardo lo si può dare all'andamento medio di ΔT tra un Context switch e l'altro. In Figure 23 e è mostrata la distribuzione dell'intervallo di tempo medio per thread di entrambi i tipi di cambio di conteso, per lo scenario con un singolo core. Si può notare che circa il 40% dei processi ha un tempo medio di ΔT sotto il 100ms.
I dati sono in linea con il comportamento di un sistema abbastanza sovraccarico, nel quale molti processi necessitano di essere schedulati e l'unica CPU disponibile deve essere arbitrata con timeslice piuttosto stretti.

La Figure 24 mostra che molti thread che sono attivi dal punto di vista della CPU, sono anche altrettanto attivi per quanto riguarda le fasi di I/O, data l'alta densità di marker nella parte più vicina all'origine degli assi in basso a sinistra.

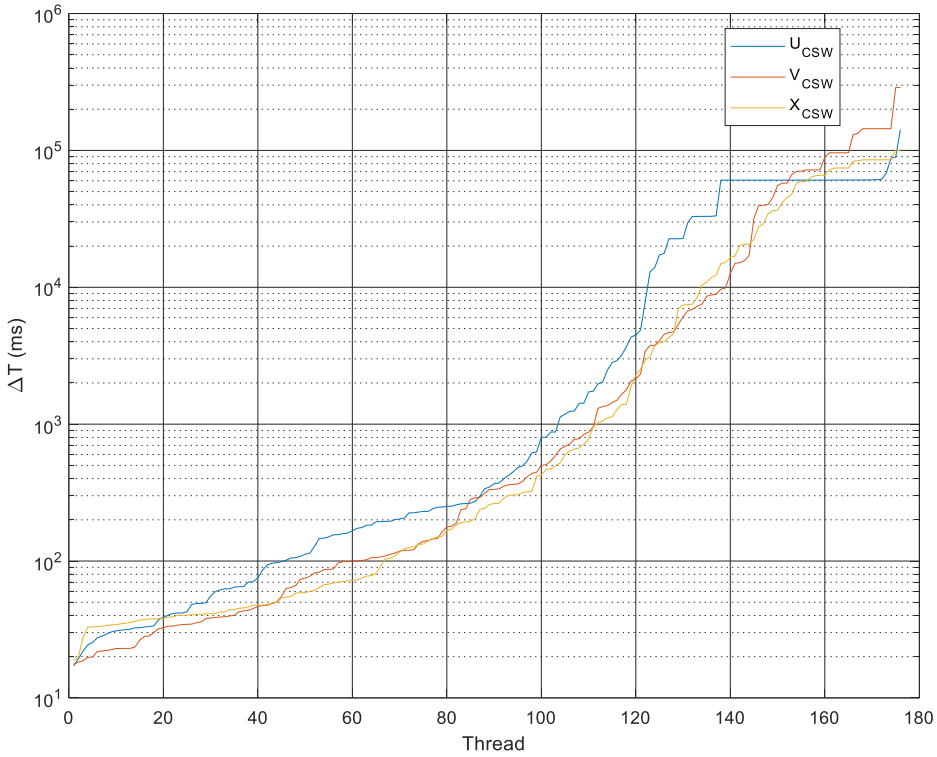

Figure 23 - Distanze temporali medie tra due context switch dello stesso tipo. Con X_{CSW} si intende la distanza media temprale tra un context switch qualsiasi.

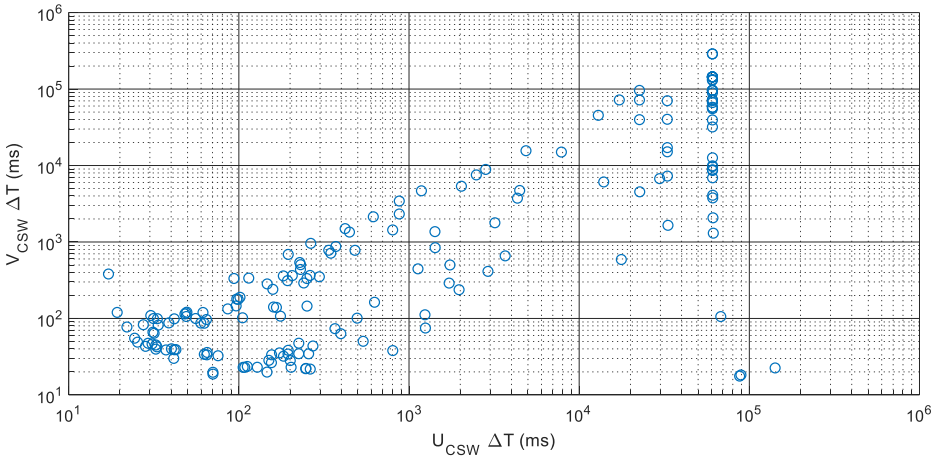

Figure 24 - Scattering delle distribuzioni temporali medie di U_{CSW} e V_{CSW} dei thread

I risultati ottenuti sul sistema a 4 cores mostrano un aumento medio della durata del timeslice per i thread che precedentemente si ammassavano sul lato in basso a sinistra di Figure 24: questo perché lo scheduler può concedere più timeslice consecutivi ai singoli thread, prima di spostarli nella coda *ready*.

Lo scenario per 4 cores è mostrato in Figure 25 e Figure 26. Il confronto tra 1 e 4 cores per quanto riguarda U_{CSW} è mostrato in Figure 27. Quello tra V_{CSW} in Figure 28.

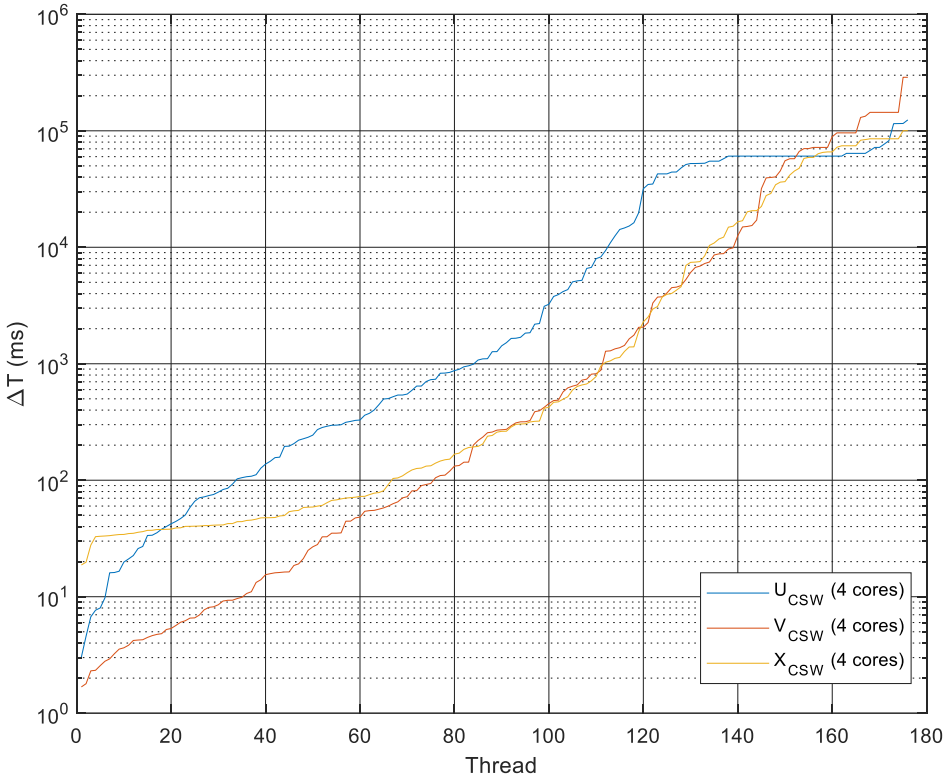

Figure 25 - Distanze temporali medie tra due context switch dello stesso tipo (4 cores). Con X_{CSW} si intende la distanza media temprale tra un context switch qualsiasi.

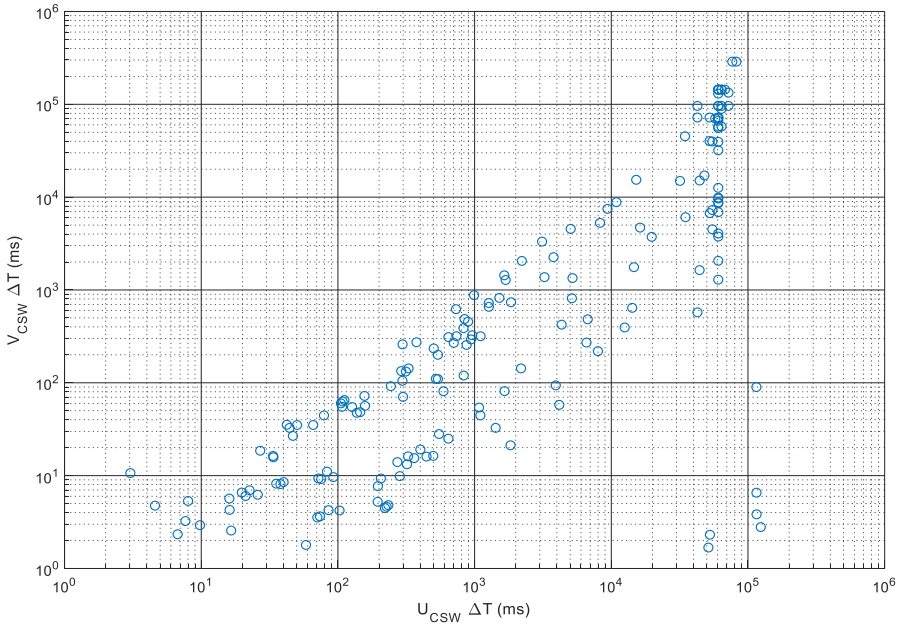

Figure 26 - Scattering delle distribuzioni temporali medie di U_{CSW} e V_{CSW} dei thread (4 cores)

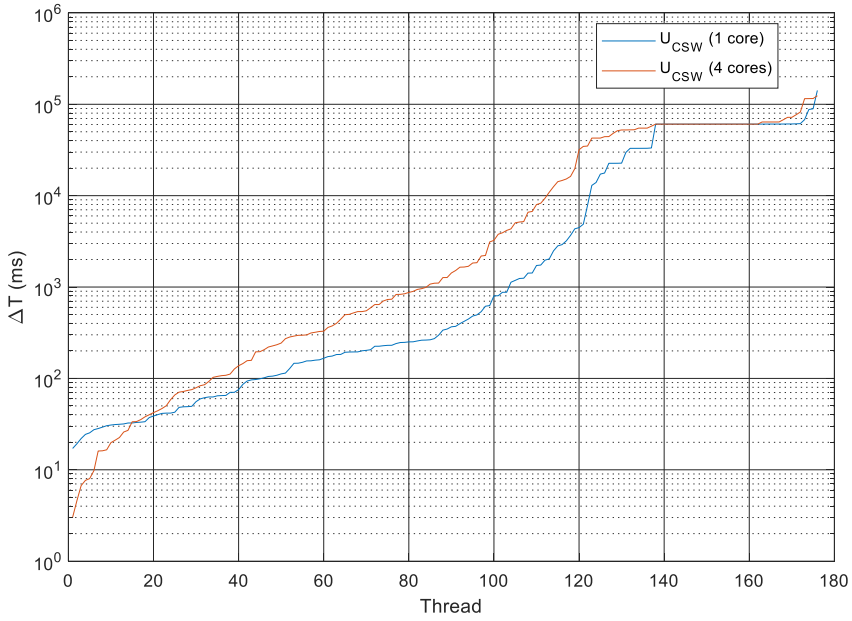

Figure 27 - Confronto tra le distanze temporali medie tra due context switch di tipo U_{CSW} (1 e 4 cores)

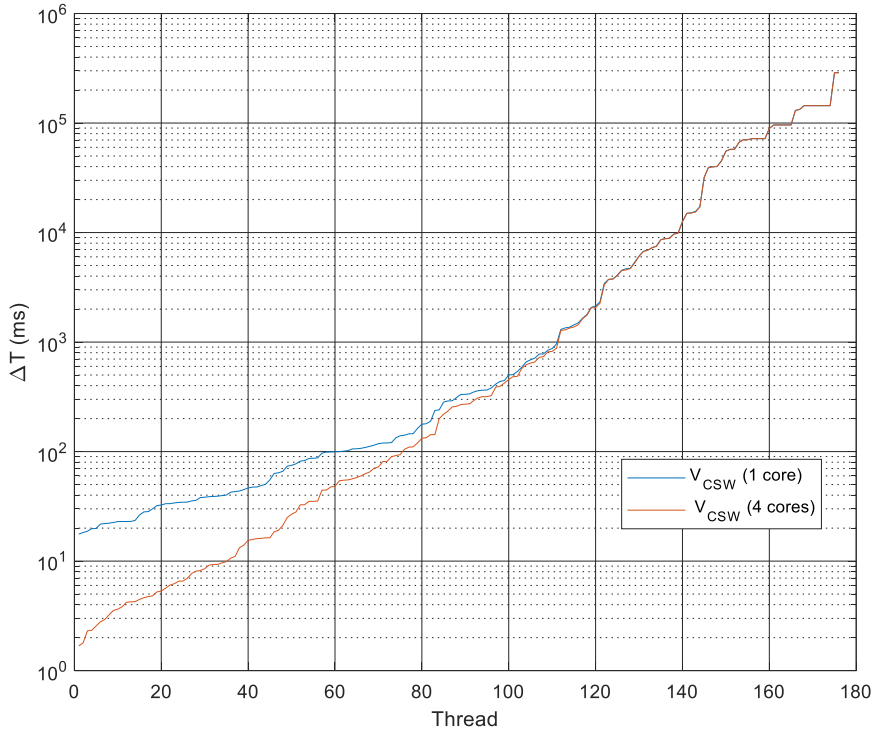

Figure 28 - Confronto tra le distanze temporali medie tra due context switch di tipo V_{CSW} (1 e 4 cores)

5.3 Distribuzione statistica delle attivazioni dei thread

Nel paragrafo 5.2 sono stati mostrati alcuni valori medi caratterizzanti i thread, come la distribuzione media degli intervalli temporali tra un context switch e il successivo. È ora necessario però focalizzarsi sull'andamento temporale delle sequenze di richiesta "esecuzione & I/O".

Lo scopo quindi è ottenere, per ogni thread, una sequenza temporale di "RUN" e "WAIT", da sottoporre ad uno scheduler semplificato, che generi la corrispondente sequenza di voluntary e unvoluntary context switch confrontabili con i dati statistici reali.

Per ottenere questo risultato, oltre al comportamento medio è necessario conoscere la distribuzione statistica dei context switch per ogni thread, in modo da poterne modellare il comportamento tramite un modello adeguato.
Un primo passo per poter modellare questo tipo di attivazioni è cercare di comprendere se esiste una qualche regolarità statistica modellabile con una distribuzione di probabilità nota.

Focalizzando temporaneamente l'attenzione su in singolo thread e studiando la distribuzione degli intervalli temporali $\Delta T(U_{CSW})$ e $\Delta T(V_{CSW})$, è possibile farsi un'idea della possibile densità di probabilità del fenomeno.

In Figure 29 e Figure 30 sono mostrate le densità di probabilità di $\Delta T(U_{CSW})$ relative ad un generico thread, come risultano dalle misurazioni effettuate con uno e quattro cores.

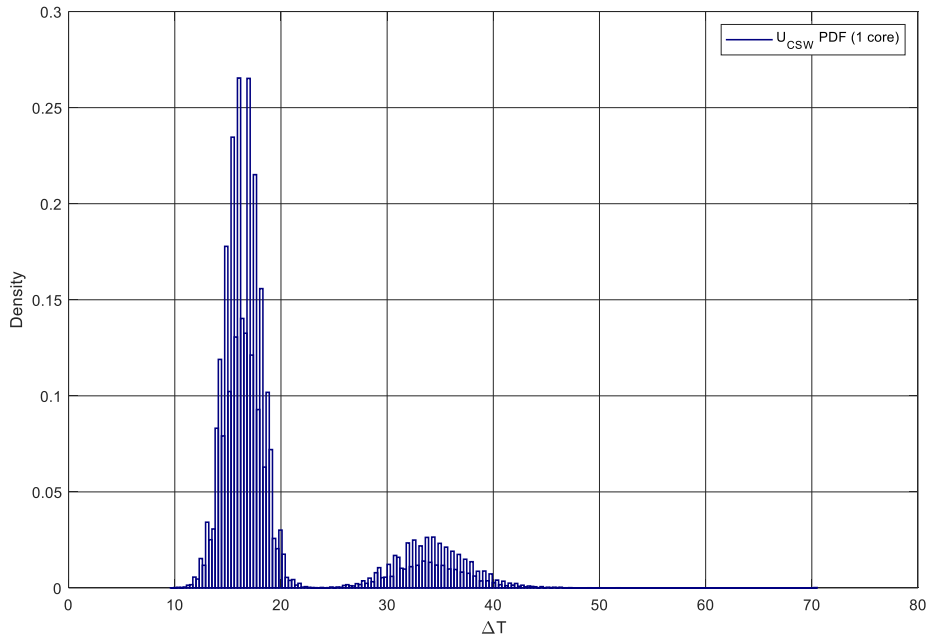

Figure 29 - Densità di probabilità di U_{CSW} per un generico thread

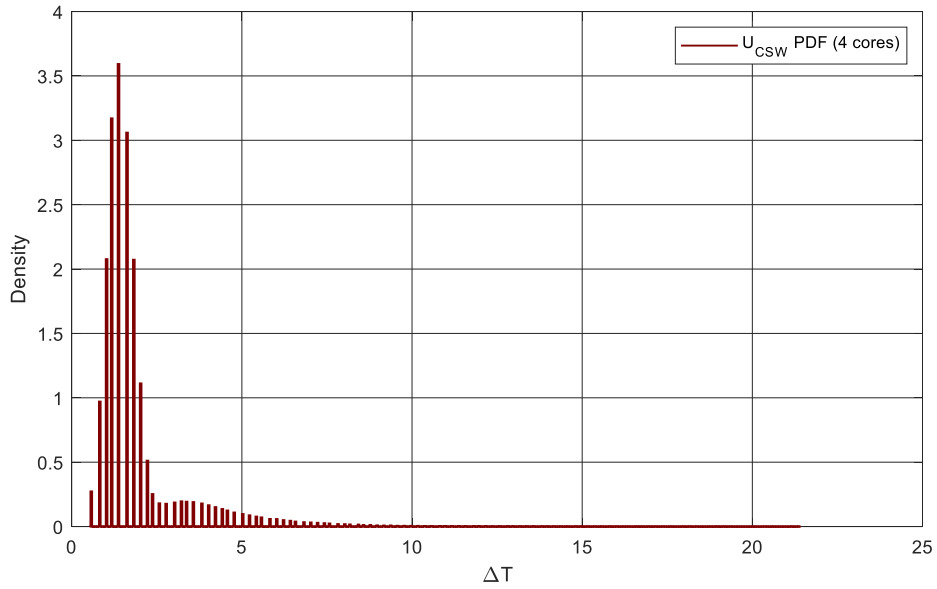

Figure 30 - Densità di probabilità di U_{CSW} per un generico thread (4 cores)

Si può notare che l'andamento della PDF non segue una distribuzione normale di probabilità, ma presenta un andamento che suggerisce un *Gaussian Mixture Model*, con almeno due gaussiane che permettano di modellare le due zone attorno alle quali si concentrano i valori.

Un modo rapido per ottenere un modello sufficientemente approssimato dell'andamento della PDF è l'utilizzo di tecniche non parametriche per la stima e la rappresentazione dei valori: la distribuzione *kernel*.
La distribuzione *kernel* è una rappresentazione non parametrica della densità di probabilità di una variabile casuale ed è definita da una funzione di *smoothing* e un valore di *bandwidth*, i quali controllano la spigolosità della curva stimata.

La funzione di *smoothing* è generalmente una distribuzione normale, mentre la *bandwidth* definisce il numero di dati sui quali effettuare la stima. Pertanto i dati vengono separati in insiemi di larghezza *bandwidth* e su di essi viene calcolata la distribuzione normale corrispondente. Il risultato della stima è una somma di gaussiane, una per ogni set di punti considerati (**EQ 26**)

Uno dei vantaggi della distribuzione *kernel* è la possibilità di evitare rigide assunzioni sulla distribuzione dei dati.
Lo stimatore kernel è definito come segue:

$$\hat{f}_h(\mathrm{x}) = \frac{1}{nh}\sum_{i=1}^{n} K\frac{x - x_i}{\mathrm{h}} \qquad \textbf{EQ 26}$$

Dove

- $x_1, \ldots x_n$ sono i valori misurati,
- n è il numero di valori disponibili
- K è la funzione di *smoothing*
- h è la *bandwidth*

In Figure 31 è visibile il risultato dello stimatore.

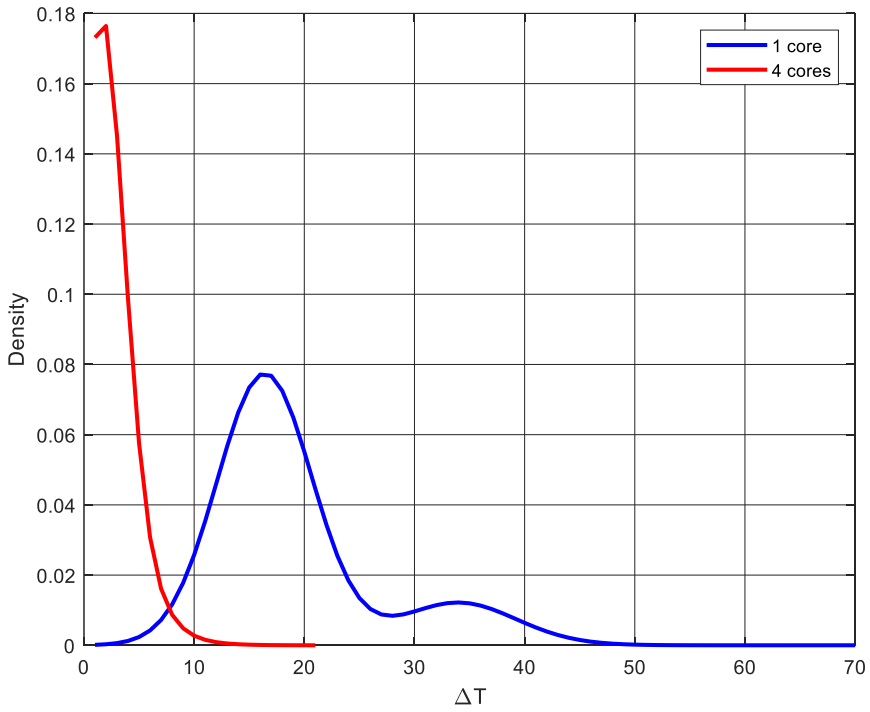

Figure 31 - Densità di probabilità (U_{CSW}) stimate tramite l'applicazione dello stimatore kernel per 1 e 4 cores.

In Figure 32, Figure 33 e Figure 34 sono mostrati i risultati ottenuti considerando i Voluntary Context Switch.

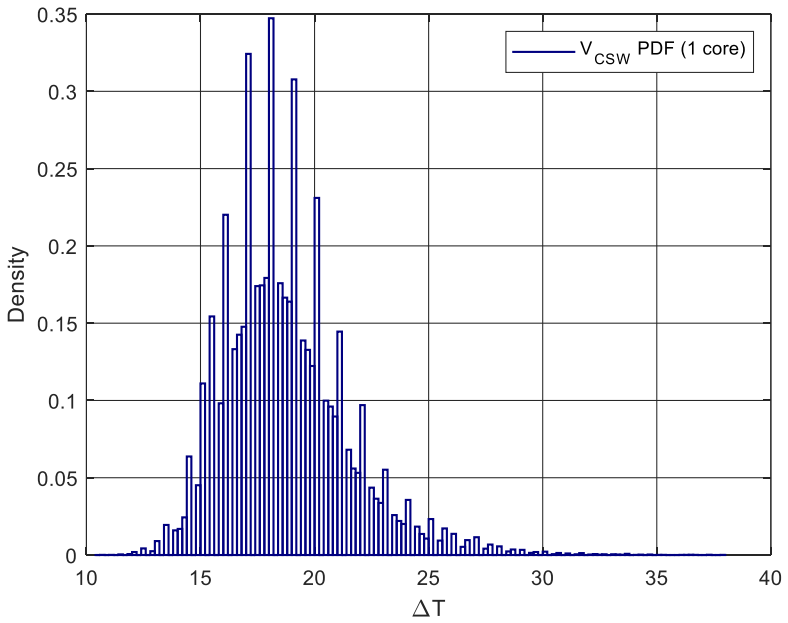

Figure 32 - Densità di probabilità di V_{CSW} per un generico thread

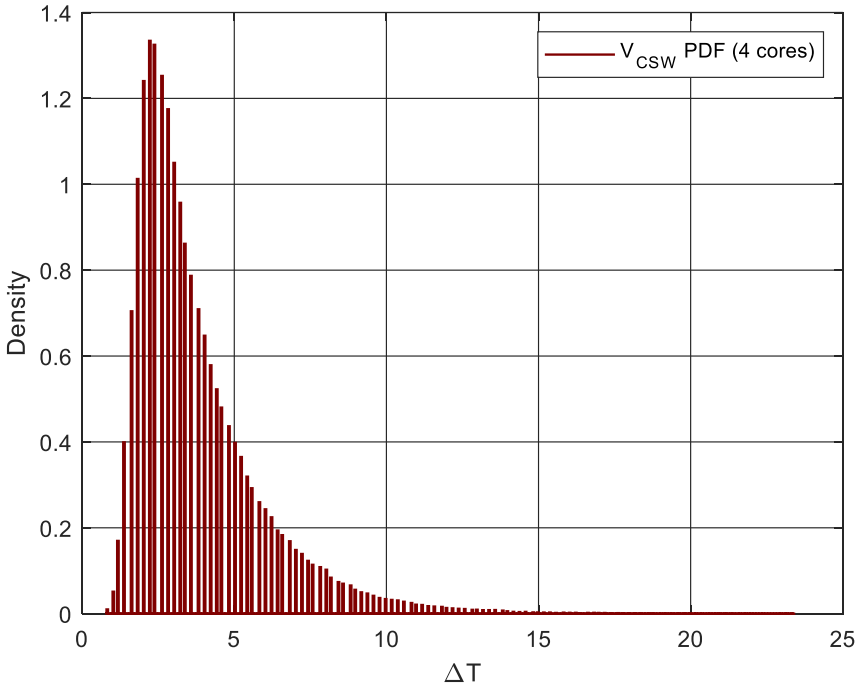

Figure 33 - Densità di probabilità di V_{CSW} per un generico thread (4 cores)

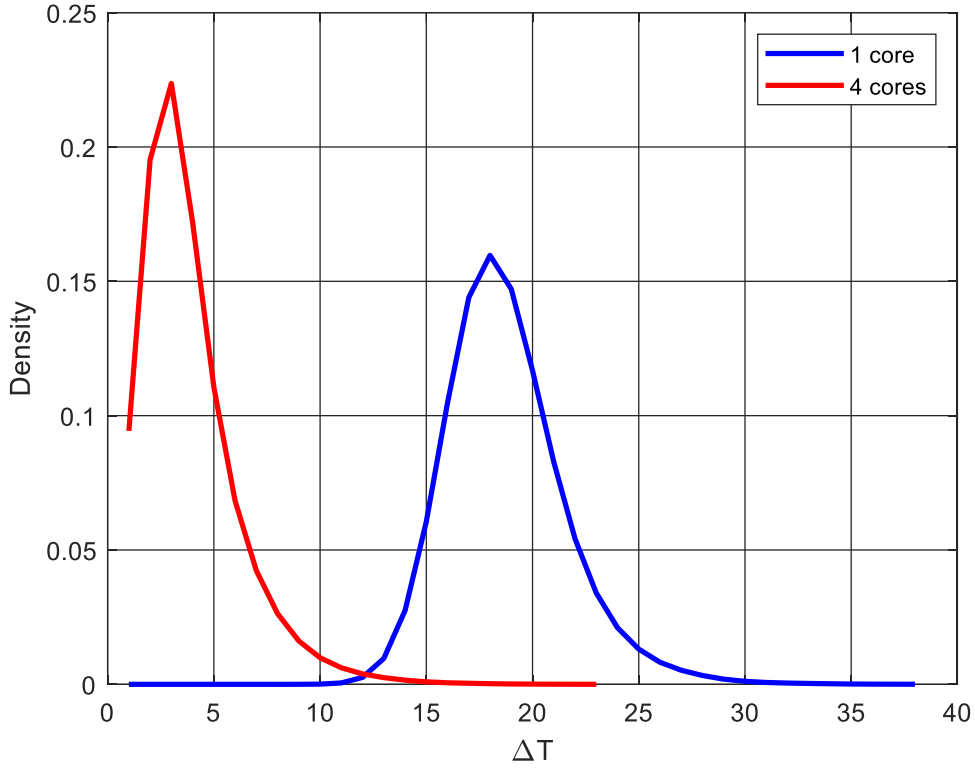

Figure 34 - Densità di probabilità (V_{CSW}) stimate tramite l'applicazione dello stimatore kernel per 1 e 4 cores.

Per verificare che la distribuzione di probabilità sia coerente con i dati di ingresso, si procede alla generazione di dati casuali con in input il *kernel* stimato, e al confronto con la sequenza misurata per determinare la corretta distribuzione.

In Figure 35 e Figure 36 sono mostrati i risultati di 3 generazioni di dati ottenuti dai modelli statistici degli U_{CSW} e V_{CSW} per un generico thread, comparati con i dati misurati.

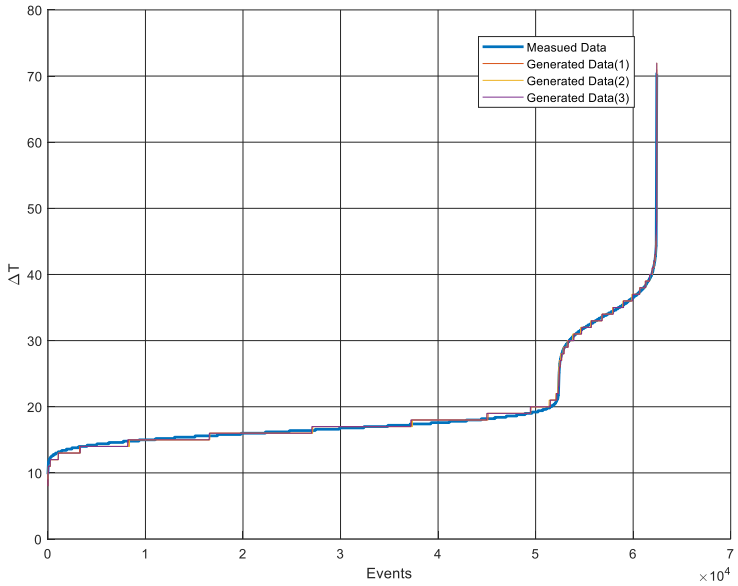

Figure 35 - Confronto tra i dati misurati e quelli generati dallo stimatore di densità di probabilità (U_{CSW}). Il confronto è stato fatto ordinando i valori in ordine crescente e disegnando la curva così ottenuta

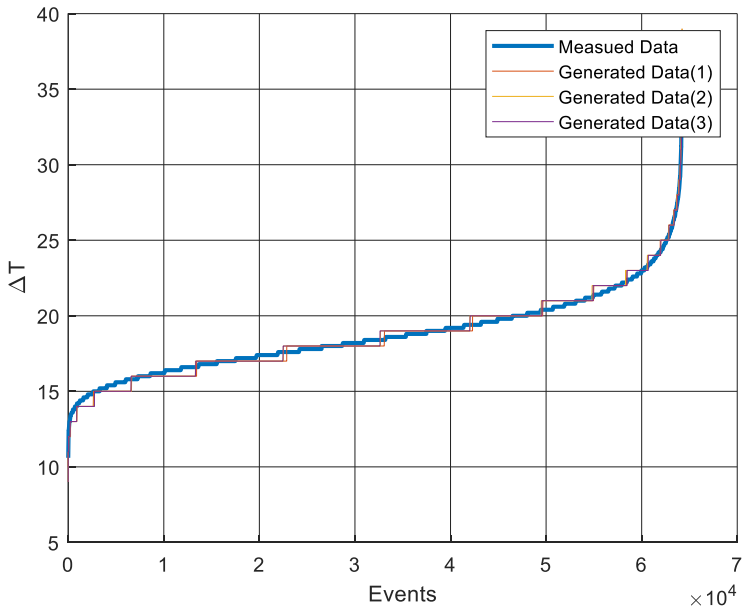

Figure 36 - Confronto tra i dati misurati e quelli generati dallo stimatore di densità di probabilità (V_{CSW})

L'analisi statistica basata sulla costruzione del modello *kernel* è stata applicata ad ogni thread presente sul sistema, utilizzando i dati osservati di voluntary e unvoluntary context switch, ottenendo 176 modelli, uno per ogni thread, che descrivono l'andamento temporale degli Unvoluntary Context Switch e altrettanti modelli che descrivono l'andamento dei Voluntary Context Switch.

Nelle Figure 37 e Figure 38 sono mostrate le distribuzioni statistiche stimate per i 176 threads.

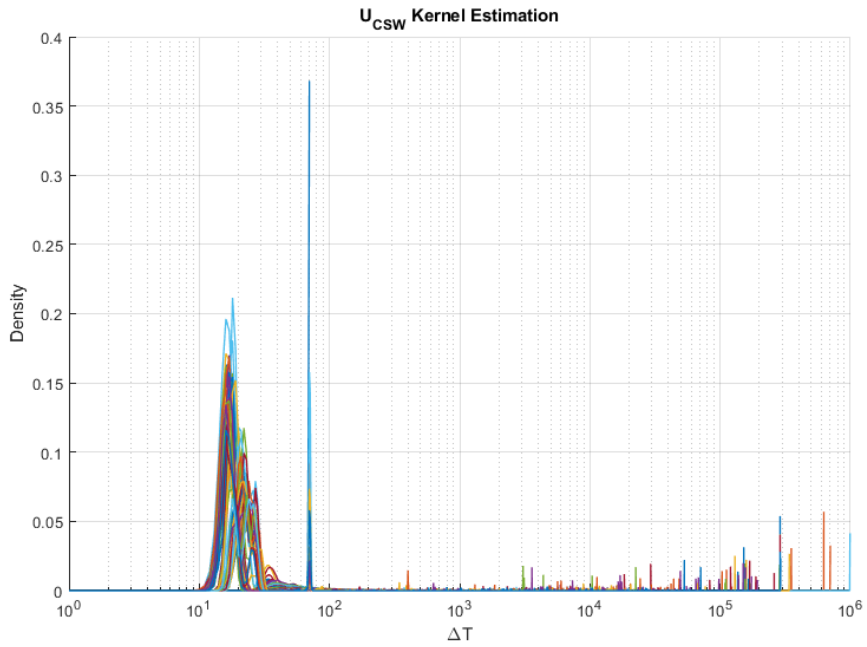

Figure 37 - Risultati degli stimatori per Ucsw applicato ai 176 thread e 1 core

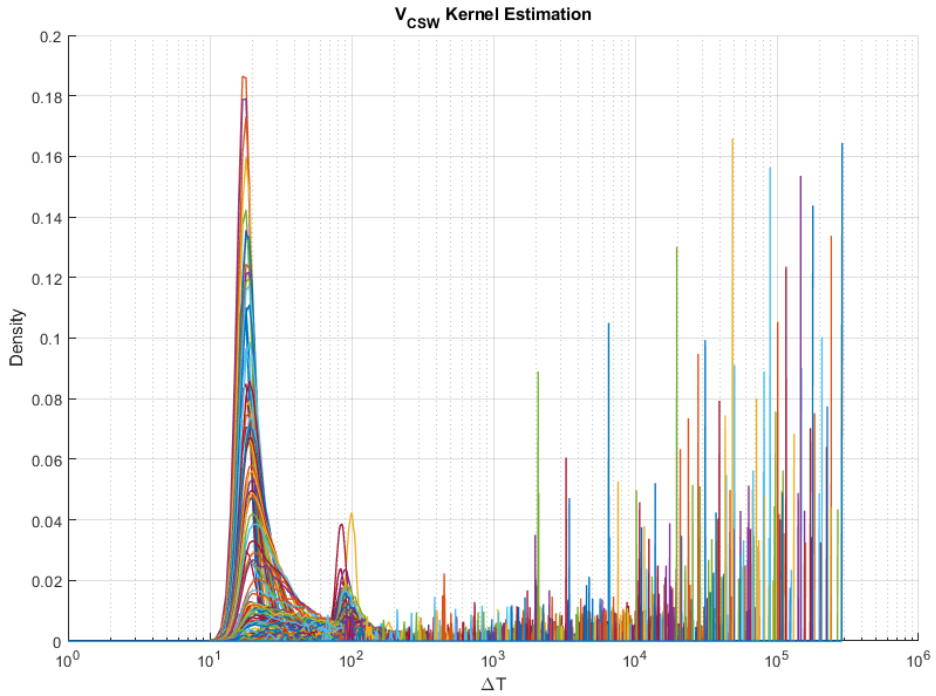

Figure 38 - Risultati degli stimatori per V_{CSW} applicato ai 176 thread e 1 core

5.4 Generazione delle sequenze RUN-WAIT

La costruzione dei modelli statistici di Unvoluntary Context Switch e Voluntary Context Switch per ogni thread è il punto di partenza per poter generare delle sequenze di stato RUN-WAIT da sottoporre allo scheduler semplificato.
Consideriamo nuovamente il significato del concetto di Voluntary e Unvoluntary;

- Voluntary Context Switch: il thread esegue una chiamata di sistema all'interno del *timeslice* assegnato, per gestire un'esigenza di I/O, di sincronizzazione, etc…
- Unvoluntary Context Switch: il timeslice associato al thread è scaduto, ma il thread, se non dovesse subire un cambio di contesto, avrebbe continuato ad seguire il proprio flusso di istruzioni.

La distanza tra un Voluntary Context Switch e il successivo sarà quindi intercalata da richieste di stati di esecuzione (RUN) e stati di attesa (WAIT) della risoluzione della causa che ha generato il cambio di contesto.
Diversamente, un context switch di tipo unvoluntary può essere modellato con una sequenza di stati RUN sufficientemente lunga da "sforare" temporalmente da durata del timeslice assegnato al thread.

La sequenza temporale di stati di RUN e di WAIT è modellata in funzione del susseguirsi di timeslices di esecuzione, considerati come unità di osservazione.
Si consideri uno slot temporale rappresentante un timeslice, dividendolo in 2 parti, e, ad ogni parte, si assegni uno stato di RUN o di WAIT.
Si potranno avere 4 diversi andamenti all'interno del timeslice, come mostrati in Figure 39.

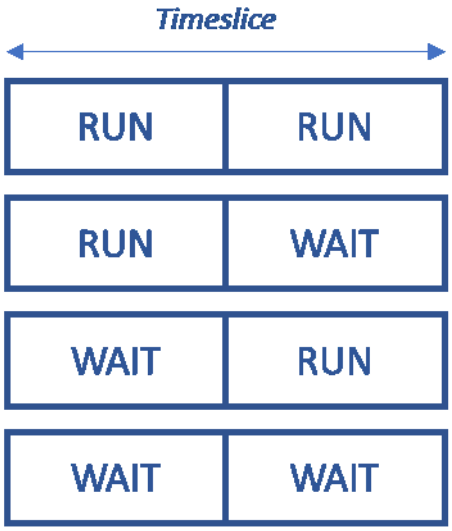

Figure 39 - I quattro tipi di distribuzione delle sequenze degli stati RUN/WAIT all'interno di un singolo timeslice

Se il timeslice sarà composto da una sequenza RUN-WAIT, si causerà, per come esso è definito, un voluntary context switch all'interno dello scheduler. Infatti esso, alla richiesta di WAIT durante il timeslice, dovrà spostare il thread in esecuzione nella

coda dei thread *waiting*. Un secondo modo per ottenere un V_{CSW} è costruire una sequenza di tipo: "[RUN-RUN] - [WAIT-X]", con X = {WAIT o RUN}.
Questa sequenza dirà allo scheduler che il primo timeslice è terminato con il thread in uno stato di "RUN", ma, nel timeslice successivo, esso richiede di passare allo stato di "WAIT", causando un V_{CSW}. Questo schema è mostrato in Figure 40.

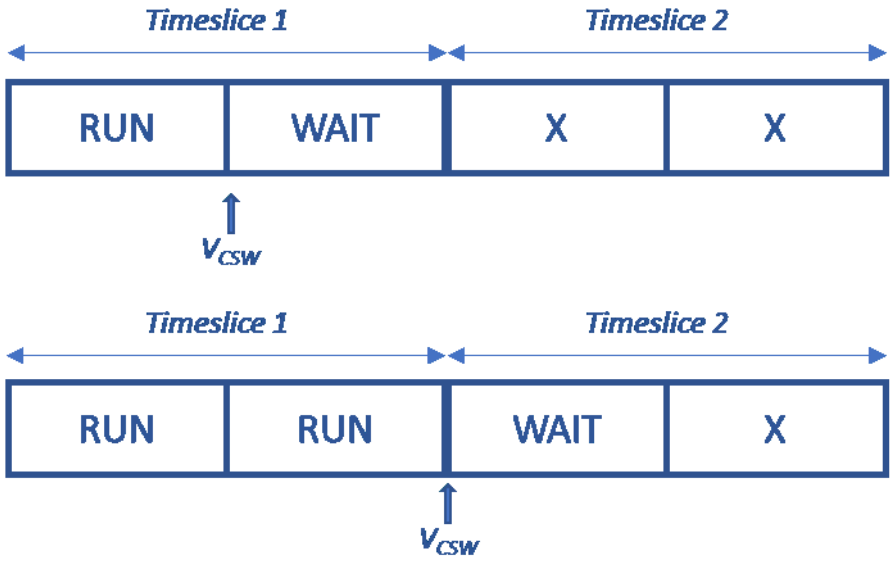

Figure 40 - Sequenze di RUN/WAIT che scatenano un V_{CSW} nello scheduler semplificato

La generazione di un unvoluntary context switch è ottenuta con una sequenza che indichi allo scheduler semplificato la volontà del thread di continuare l'esecuzione oltre la durata del timeslice. Per ottenere questo risultato basta considerare una singola sequenza dove il numero di RUN consecutivi sarà superiore a 2: "[RUN-RUN] - [RUN-X]" con X = {WAIT o RUN}. La sequenza è mostrata in Figure 41.

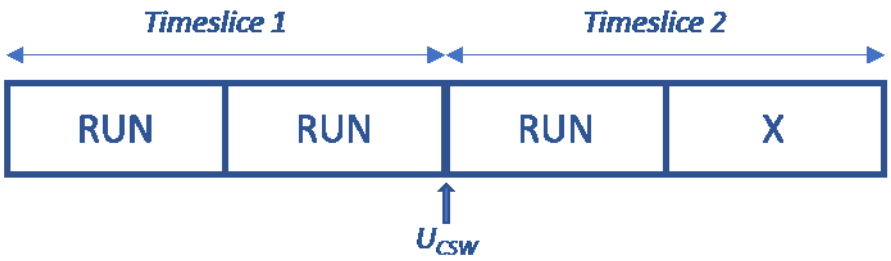

Figure 41 - Sequenza di RUN che scatena un U_{CSW} nello scheduler semplificato

Una sequenza prolungata di stati di WAIT successivi al V_{CSW} o U_{CSW} permette di modellare la distanza temporale tra un Context Switch e il successivo, indipendentemente dalla sua tipologia.

In Figure 42 è mostrato un esempio di sequenza temporale, evidenziando i $\Delta U_{CSW}(ts)$ e $\Delta V_{CSW}(ts)$.

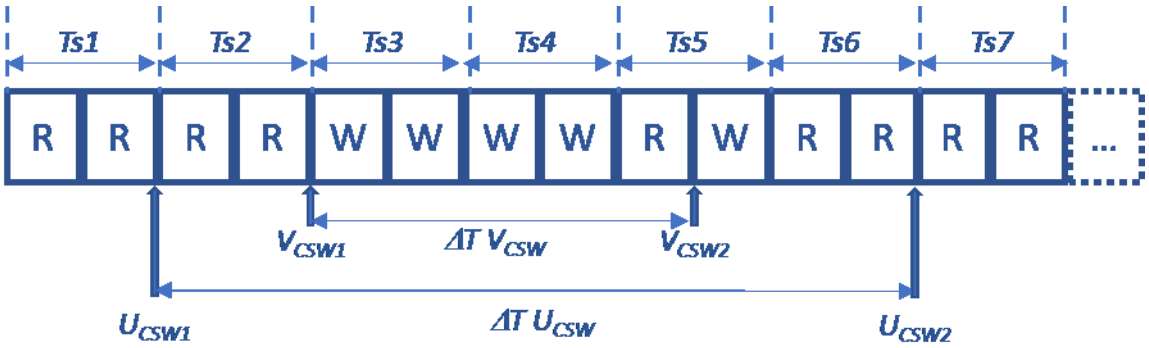

Figure 42 - Esempio di schema temporale causante i context switch U_{CSW} e V_{CSW}, e i relativi ΔU_{CSW} (ts) e ΔV_{CSW} (ts)

Utilizzando i modelli statistici delle distanze tra voluntary e unvoluntary context switch misurati nello scenario con un singolo core, si procede, per ogni thread, alla creazione di una linea temporale di eventi di context switch, conformemente alla distribuzione di probabilità di distanze temporali.

Le varie sequenze vengono quindi tradotte in sequenze di stati "RUN/WAIT" utilizzando le regole descritte.

Chiaramente, un context switch generato dalla sequenza coinciderà o meno con un context switch nello scheduler semplificato, in funzione del numero di cores che lo scheduler avrà a disposizione.

Le sequenze così ottenute sono state "eseguite" dallo scheduler semplificato, e l'andamento dei Voluntary e Unvoluntary Context Switch per ogni thread è stato confrontato con i dati reali misurati, in funzione del numero di cores.

5.5 Algoritmo di generazione delle sequenze RUN/WAIT

Come introdotto nel paragrafo 5.4 la generazione delle sequenze di RUN/WAIT per ogni thread è legata alla statistica della frequenza temporale dei voluntary e unvoluntary context switch.
Il primo passo per ottenere le sequenze richieste è definire un tempo di simulazione e, conseguentemente, la corrispondenza in *timeslices*. In un sistema Linux con scheduler di tipo CFS la durata del timeslice non è predefinita e rigida ma varia tra i valori *sysctl_sched_min_granularity* (default 0.75ms) e *sysctl_sched_latency* (default 6ms) con un target di default pari a 20ms complessivi per tutti i thread, e un incremento a blocchi di 4ms. Per rendere il problema trattabile si definisce, in prima approssimazione, un timeslice costante pari a 10ms. Un' ora di simulazione coinciderà con 360.000 timeslices per core.

Si considerino, per ogni thread I, i seguenti elementi:

- ΔTIME la durata in ms del timeslice;
- D = durata della simulazione (ad es. 1h = 3600000ms);
- N = D/ΔTIME numero di timeslices contenute nella durata della simulazione
- *kernelU*$_{CSW}$*Thread*(I) il modello statistico delle distanze degli unvoluntary context switch;
- *kernelV*$_{CSW}$*Thread*(I) il modello statistico delle distanze dei voluntary context switch;
- *X*$_{CSW}$*Array*(I, N) l'array contenente la timeline temporale del thread, inizializzata a 0 (nessun context switch)

L'algoritmo per ottenere le sequenze RUN/WAIT per il thread I considera inizialmente gli Unvoluntary Context Switch ed è composto dai seguenti passi:

1. *ArrayPos* = 1;
2. *X*$_{CSW}$*Array*(I, *ArrayPosition*) = U$_{CSW}$
3. *ArrayPos* += round(random(*kernelU*$_{CSW}$*Thread*(I))/ΔTIME);

4. $X_{CSW}Array$(I, *ArrayPosition*) = U_{CSW}
5. WHILE *ArrayPosition* <= N GOTO 3

Nel dettaglio:

- i passi 1 e 2 impongono un unvoluntary context switch di "boot" nella prima posizione utile;
- il passo 3 calcola, utilizzando il modello statistico delle distanze degli unvoluntary context switch, la prossima posizione dove deve comparire il prossimo context switch;
- il passo 4 impone un unvoluntary context switch nella posizione calcolata;
- il passo 5 rimanda ai passi 3 e 4 fino a coprire l'intero arco temporale della simulazione.

Lo stesso processo viene seguito per i Voluntary Context Switch, con alcune differenze:

1. *ArrayPos* = 1;
2. *Next* = round(random($kernelV_{CSW}Thread$(I)) /ΔTIME);
3. IF *ArrayPos* + *Next* <= N AND $X_{CSW}Array$(I,*ArrayPos*)==0
    - *ArrayPos* += *Next*;
    - $X_{CSW}Array$(I,*ArrayPos*) = V_{CSW}
4. WHILE *ArrayPosition* <= N GOTO 2

Nel dettaglio:

- il passo 1 inizializza l'indice dell'array a 1;
- il passo 2 calcola, utilizzando il modello statistico delle distanze dei voluntary context switch, la prossima posizione dove deve comparire il prossimo;
- il passo 3 impone un voluntary context switch nella posizione calcolata solo se il timeslice considerato non è già stato assegnato ad un unvoluntary context switch;

- il passo 4 rimanda ai passi 2 e 3 fino a coprire l'intero arco temporale della simulazione.

Al termine della procedura, per ogni thread si ha una sequenza di U_{CSW} e V_{CSW} coerenti con le rispettive distribuzioni di probabilità.

Per poter ottenere le sequenze finali è possibile verificare se, per ogni timeslice J (J diverso da 1), vi sono più thread che richiedono un context switch (Figure 43).

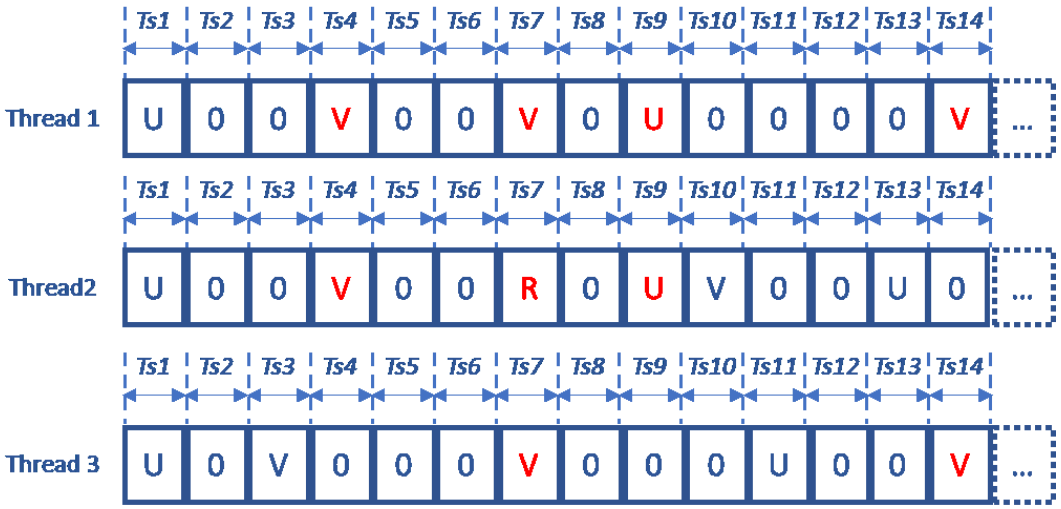

Figure 43 – Sequenze temporali di context switch e possibili conflitti (in rosso)

Il numero massimo di context switch sovrapposti sullo stesso timeslices viene limitata al numero di cores considerati.

In caso di conflitti, si procede come segue:

1. si ordinano, in ordine crescente, i thread in base alla distanza del context switch precedente o successivo dello stesso tipo;
2. si escludono i primi K processi (K uguale al n. di cores considerati)
3. Per i restanti thread I, si sposta il context switch nella più vicina timeslice libera, precedente o successiva, che non generi conflitto con i thread precedenti [I-1, I-2, …, K+1].

La scelta di ordinare i thread per distanze crescenti dall'ultimo context switch permette di applicare gli spostamenti in proporzione alle distanze.

Le sequenze finali {U, V, 0}, così ottenute, vengono convertite in sequenze RUN/WAIT seguendo le regole descritte nel paragrafo 5.4.
Al valore 0 è assegnata la sequenza [WAIT-WAIT].

5.6 Lo scheduler semplificato

Lo studio delle sequenze di RUN/WAIT per ogni singolo thread, e la loro interazione reciproca, necessita di un tool in grado di interpretare le sequenze e di tenere traccia degli eventi di Voluntary e Unvoluntary context switch.
Il tool è stato sviluppato in C# e, tramite file di configurazione in formato json, riceve in input il numero di cores disponibili per lo scheduler e le sequenze dei thread da gestire.
In output vengono salvati gli eventi di Voluntary e Unvoluntary Context switch, accompagnati da un timestamp, generati da ogni singolo thread; più una serie di statistiche globali.

Lo scheduler esegue le sequenze a passi temporali di Timeslice/2, allocando i thread ai cores o alle code di attesa in funzione dello stato dei thread stessi.

Vengono gestite 3 code distinte:

- coda dei thread *ready*: thread non in esecuzione ma che presentano come primo stato utile lo stato "RUN";
- coda dei thread *waiting*: thread non in esecuzione ma che presentano come primo stato utile lo stato "WAIT";
- coda dei processi in esecuzione.

Ad ogni loop avvengono i seguenti passi:

- i thread nella coda *waiting* avanzano di un passo nella sequenza; se lo stato successivo è ancora WAIT, vengono rimessi in coda, se lo stato successivo è RUN, vengono accodati alla coda *ready*
- se esiste un core libero (è stato eseguito un context switch dal thread che lo occupa attualmente), viene scelto il primo thread della coda *ready* come prossimo thread da eseguire dal core;
- per ogni core gestito dallo scheduler:
    - se lo stato del thread associato è pari a RUN e *CurrentTimeslice* = Timeslice/2, lo stato viene "consumato" e il thread resta associato al core;
    - se lo stato del thread è RUN e *CurrentTimeslice* = Timeslice, lo stato viene "consumato" ma viene verificato lo stato successivo: se è ancora RUN, avviene un context switch di tipo Unvoluntary sul thread in esecuzione, ed esso viene accodato nella coda *Ready*, se lo stato futuro è WAIT, avviene un context switch di tipo Voluntary sul thread ed esso viene accodato alla coda *Waiting;*
    - Se la coda dei processi *Ready* è vuota, il context switch di tipo Unvoluntary non avviene e il thread resta associato al core;
    - Il context switch di tipo Voluntary avviene in ogni caso: se non vi sono thread nella coda *ready* da allocare al core, esso resta in stato di "idle".
    - In presenza di un Context Switch, il valore di CurrentTimeslice viene azzerato, altrimenti viene incrementato di Timeslice/2.

Come descritto nel paragrafo 5.4, il valore di Timeslice è convenzionalmente definito pari a 2.

5.7 Prestazioni del simulatore

L'approccio proposto è stato validato generando una serie di insiemi di sequenze RUN/WAIT per i 176 thread ed eseguendo il simulatore di sistema, basato sullo scheduler semplificato.
I dati di voluntary e unvoluntary context switch sono stati quindi confrontati con i valori misurati sul sistema reale, in modo da valutare eventuali scostamenti.

Nella Figure 44 sono mostrati gli andamenti del numero di Unvoluntary Context switch per ogni thread ottenuti con il simulatore di sistema, a confronto con i dati misurati.

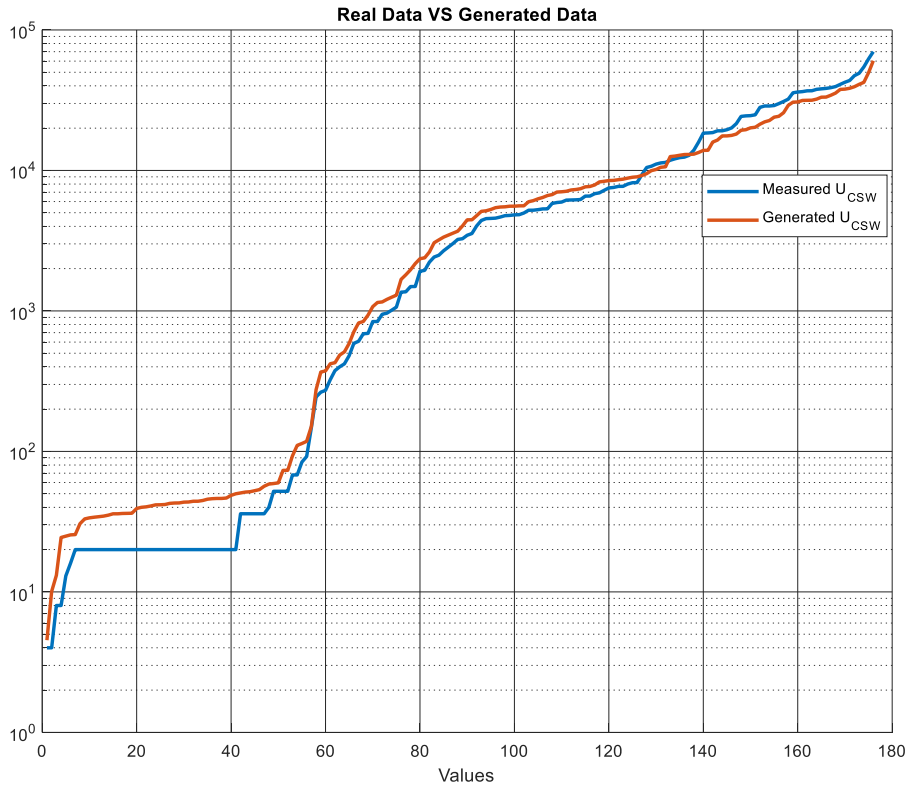

Figure 44 - Unvoluntary Context Switch generati dal simulatore utilizzando le sequenze RUN/WAIT generate statisticamente per ogni thread, confrontate con i dati acquisiti tramite misure sul sistema reale

Si può notare che la sequenza creata artificialmente genera un numero di Unvoluntary Context Switch leggermente sottostimato quando essi sono in numero ridotto (in basso a sinistra), mentre presenta un leggera tendenza alla sovrastima per valori più elevati (parte in altro a destra del grafico). Questo andamento è confermato dalla CDF di Figure 45.

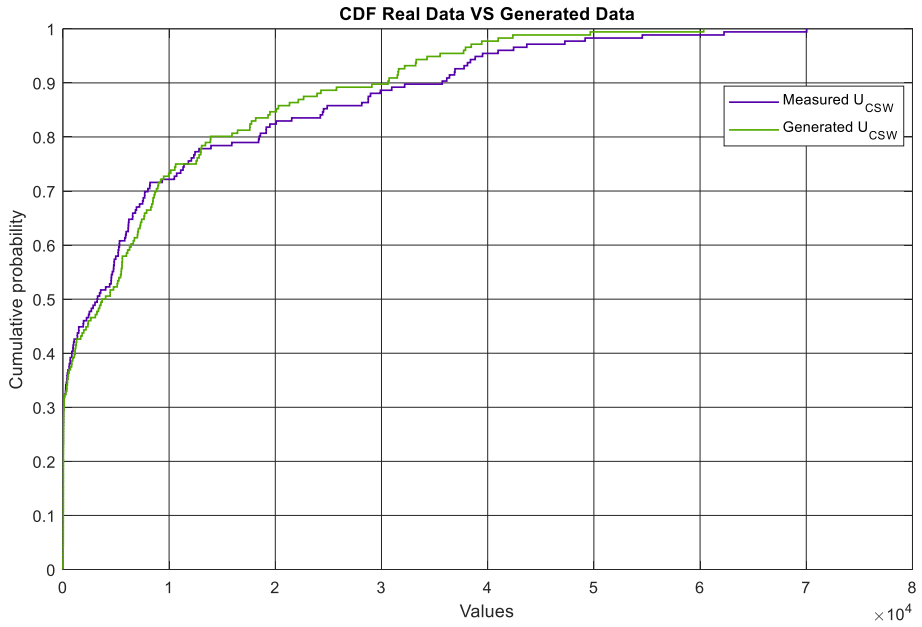

Figure 45 - CDF degli U_{CSW} generati confrontati con i U_{CSW} misurati

In Figure 46 sono mostrati i dati riguardanti i Voluntary Context Switch generati artificialmente, confrontati con i dati misurati.
Si può notare, per questo tipo di context switch, una lieve tendenza alla sovrastima, benché limitata, su tutto il range di valori.

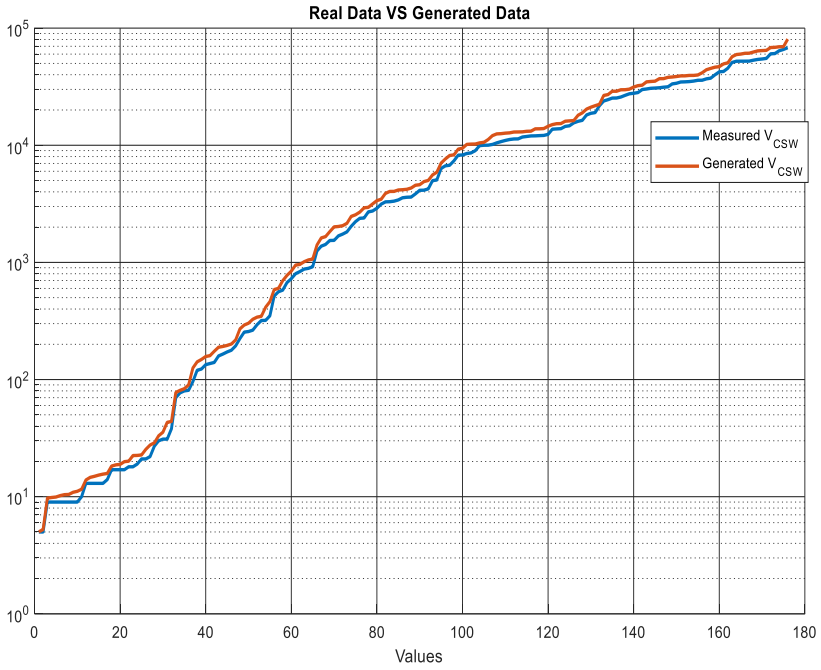

Figure 46 - Voluntary Context Switch generati dal simulatore utilizzando le sequenze RUN/WAIT generate statisticamente per ogni thread, confrontate con i dati acquisiti tramite misure sul sistema reale

Anche in questo caso, l'andamento della CDF risultante conferma le osservazioni, come mostrato in Figure 47.

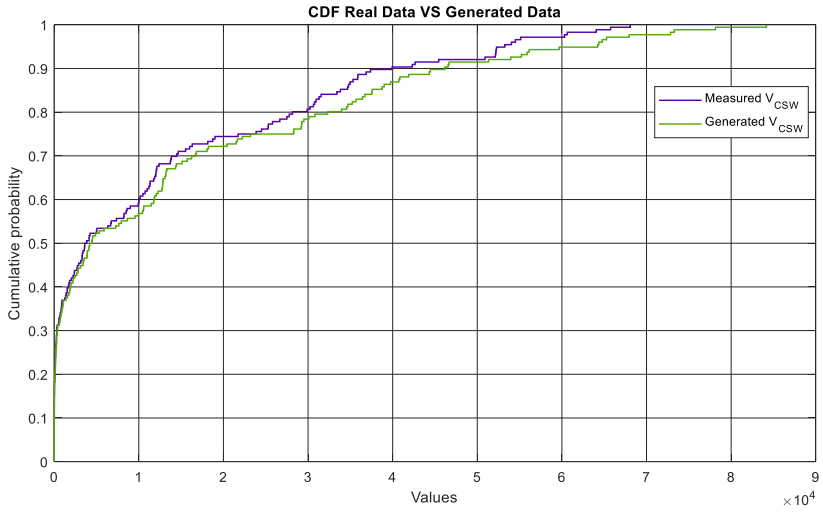

Figure 47 - CDF degli V_{CSW} generati confrontati con i V_{CSW} misurati

I dati U_{CSW} e V_{CSW} possono ora essere applicati al modello proposto per verificare se l'andamento predittivo del modello è confrontabile con i risultati ottenuti basandosi sui dati misurati.

Allo scopo, sono stati rieseguiti i passi e i calcoli relativi per ottenere i valori di TP(Generated) e SOr(Generated) come discusso nei Capitoli 3 e 4, proiettando la stima di incremento delle prestazioni da un core fino a 16 cores.
I risultati ottenuti e la loro comparazione con i dati precedentemente calcolati sono mostrati in Figure 48.

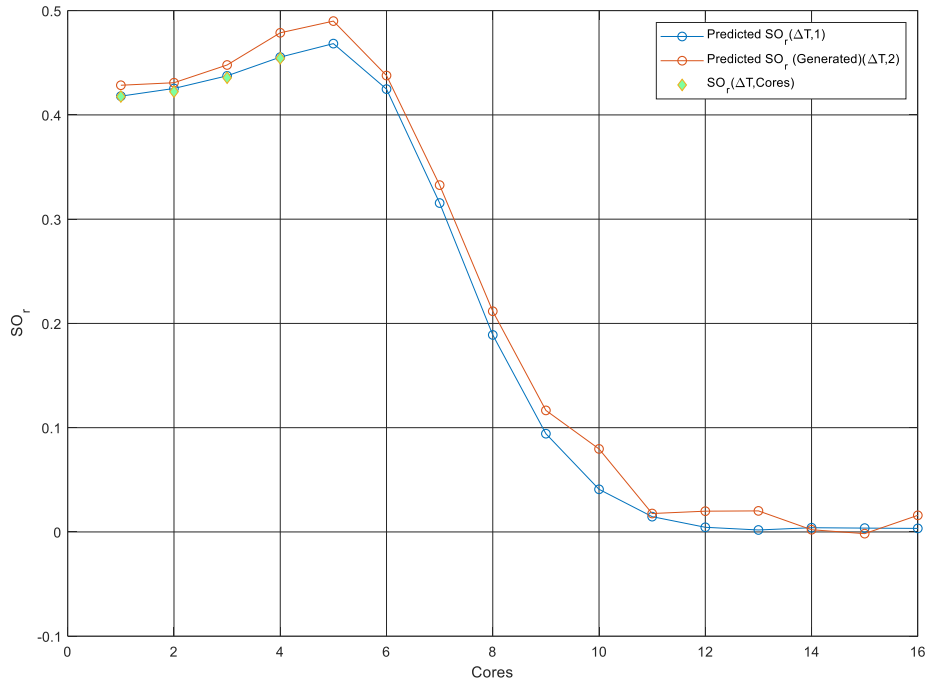

Figure 48 – Confronto tra l'andamento della predizione di SOr(ΔT) ottenuta con i dati misurati e con i dati generati attraverso la generazione statistica delle sequenze RUN/WAIT e l'esecuzione dello scheduler semplificato

L'andamento della curva di SOr(Generated,ΔT) mostra un andamento qualitativo confrontabile con i valori di SOr(ΔT).
In generale si può notare una lieve sovrastima dei valori calcolati, probabilmente imputabili alla corrispondente sovrastima degli Unvoluntary Context Switch mostrata in Figure 44.

I risultati ottenuti confermano la possibilità di modellare statisticamente il comportamento dei thread e, di conseguenza, generare i corrispettivi eventi di Voluntary e Unvoluntary Context Switch tramite la simulazione della loro esecuzione attraverso uno scheduler semplificato. Questa evidenza sperimentale incoraggia ad utilizzare questo metodo per verificare, tramite l'aggiunta ragionata di pacchetti di thread con determinati livelli di "CPU-Bound" e "I/O-Bound", l'impatto sul sistema, in termini di prestazioni, aggiungendo una nuova funzionalità, determinando di conseguenza se sia vantaggioso introdurre un incremento di capacità computazionale e stimandone la quantità più adeguata.

La strategia proposta trova pertanto applicazione immediata negli scenari che un progettista di sistemi infotainment si trova oggi ad affrontare: la convergenza di una sempre maggiore quantità di funzionalità in una singola centralina; la necessaria valutazione dell'effettivo utilizzo da parte della componente software delle risorse di sistema messe a disposizione.
Specularmente, l'approccio proposto può essere un valido aiuto per valutare il trade-off costi/benefici di un eventuale *upgrade* del sistema hardware, pianificabili nelle nuove varianti di prodotti attualmente in produzione a fronte degli inevitabili aggiornamenti software sviluppati per soddisfare le nuove esigenze di impiego del sistema installato in vettura da parte degli utenti.

Capitolo 6

Conclusioni

6.1 Conclusioni generali

Il dimensionamento di un sistema, di qualsiasi natura, che rispetti i requisiti richiesti al minor costo possibile è una sfida ardua. La sfida si complica ulteriormente se le condizioni da soddisfare variano rapidamente, sia durante il design e lo sviluppo del sistema, sia durante il suo ciclo di vita, fino alla conclusione del periodo di produzione. Il mondo automotive, storicamente meno incline all'introduzione di cambiamenti nel breve termine rispetto ad altri settori considerati più dinamici, sta sperimentando una evoluzione tecnologica sempre più veloce anno dopo anno, e di conseguenza una concorrenza tra i player del settore.

È pertanto sempre più pressante la necessità di disporre di strumenti, sia analitici, sia semi-empirici per predire le prestazioni del sistema dato una configurazione hardware/software e ottimizzare i costi sia di sviluppo, sia di *bill of material* (BOM).

Lo scopo di questo lavoro consiste nella presentazione di un modello predittivo che permetta di dedurre e verificare il giusto mix di potenza computazionale che una suite software, eseguita su un sistema *infotainment* automotive, deve disporre per garantire un certo livello di prestazioni.

Il modello proposto si basa su una nuova formulazione dell'espressione di incremento prestazionale a fronte dell'aggiunta di una risorsa di calcolo addizionale; questa formulazione prevede come fattori chiave il numero di Voluntary e Unvoluntary Context Switch (V_{CSW} e U_{CSW}) che ogni thread subisce durante l'esecuzione in una finestra temporale di osservazione. L'espressione che lega i V_{CSW} e U_{CSW} di ogni thread definisce una grandezza chiamata *Thread Parallelization Ratio* (TPr). Questo parametro dà conto, per ogni thread, del suo livello di "CPU-Bound" e "I/O-Bound", relativo a tutti i thread presenti nel sistema.

La media pesata dei valori TPr di ogni thread sul loro numero totale prende il nome di *System Overbooking Ratio* (SOr). Questo valore rappresenta complessivamente la parte potenziale del software che potrebbe giovare di una risorsa computazionale aggiuntiva, in similitudine all'effetto complessivo del parametro *f* della legge di Amdahl, come descritta nell'Appendice A, applicato ad ogni singolo thread.

La predizione dell'incremento prestazionale S(N) viene quindi ottenuta applicando la legge di Amdahl ad ogni thread, utilizzando il parametro $TPr(thread_i)$ e riscalando il numero di U_{CSW} del valore ottenuto dalla valutazione di S(N). Il nuovo valore di System Overbooking Ratio SOr(N) è ottenuto ricalcolando i nuovi valori di TPr(N) e ottenendone la media pesata sul numero di thread.

La validazione del modello proposto si basa sull' acquisizione di dati di context switch misurati durante alcune sessioni di test utilizzando un sistema infotainment reale, variando il numero di CPU (cores) disponibili per l'esecuzione del software e confrontando i valori di TPr e SOr calcolati nel caso reale e predetti dal modello, al variare del numero di cores. Il sistema è stato utilizzato in modo tale da attivare una

serie di use cases significativi a produrre un carico di lavoro compatibile con l'uso che ne farebbe l'utente finale.

I risultati ottenuti mostrano che il modello proposto è in grado di predire correttamente l'andamento dei suddetti valori e il trend generale del vantaggio computazionale atteso, dando chiare indicazioni del numero di cores necessari a saturare la capacità potenziale del software di giovarsi di una risorsa computazionale aggiuntiva.

La raccolta dei dati sul campo è generalmente dispendiosa, sia in termini di tempo, sia in costi di esercizio poiché ogni sessione richiede l'esecuzione del software su un sistema installato su un veicolo reale, alcune modifiche del kernel per la raccolta dei dati, il coinvolgimento di drivers e utenti che utilizzino fisicamente il sistema e la verifica che i più importanti use cases di interesse siano effettivamente eseguiti.
È risultato subito chiaro che sarebbe stato molto utile poter disporre di un tool in grado di generare profili di U_{CSW} e V_{CSW} del singolo thread con un approccio statistico conforme alla suite del software sotto analisi.

In questo lavoro è stato proposto un approccio statistico al problema, definendo un metodo di generazione di sequenze di "RUN-WAIT" rappresentanti, per ogni thread, le richieste di accesso o meno alla CPU da parte dei singoli thread.
Queste sequenze sono state quindi fatte interpretare da uno scheduler semplificato sviluppato in C# il quale, alla fine di ogni sessione di esecuzione, calcola, per ogni thread, le rispettive sequenze di U_{CSW} e V_{CSW}.

I risultati ottenuti e il loro confronto con i risultati ottenuti tramite l'analisi dei dati raccolti sul sistema reale, confermano la possibilità di modellare statisticamente il comportamento dei thread costituenti la suite software e, di conseguenza, generare i corrispettivi eventi di Voluntary e Unvoluntary Context Switch tramite la simulazione della loro esecuzione attraverso lo scheduler semplificato. Questa evidenza sperimentale incoraggia ad utilizzare questo metodo per verificare, tramite l'aggiunta ragionata di pacchetti di thread con determinati livelli di "CPU-Bound" e "I/O-

Bound", l'impatto sul sistema, in termini di prestazioni, aggiungendo una nuova funzionalità, determinando di conseguenza se sia vantaggioso introdurre un incremento di capacità computazionale e stimandone la quantità più adeguata.

La strategia proposta trova pertanto applicazione immediata negli scenari che un progettista di sistemi infotainment si trova oggi ad affrontare: la convergenza di una sempre maggiore quantità di funzionalità in una singola centralina; la necessaria valutazione dell'effettivo utilizzo da parte della componente software delle risorse di sistema messe a disposizione.

6.2 Sviluppi futuri

Il modello di predizione proposto, insieme al metodo di generazione statistica ed esecuzione dei thread attraverso uno scheduler semplificato è stato messo alla prova principalmente su un sistema infotainment quad-core, al quale sono stati attivati tutti i cores disponibili, o solo una parte di essi.
I futuri prodotti prevedono l'impiego di system-on-chip con otto o più cores omogenei, in aggiunta ad eventuali altri cores di tipo diverso e le suite software prevederanno l'utilizzo pervasivo di tecnologie di virtualizzazione dell'hardware (*hypervisor*) per abilitare la coesistenza e l'esecuzione di diversi sistemi operativi e funzionalità eterogenee.

Sarà pertanto possibile, nel breve periodo, convalidare i risultati ottenuti applicando questa metodologia in nuovi scenari, dove la massiccia integrazione di diversi domini funzionali richiederà ancora maggiore efficacia nell'allocazione e al dimensionamento delle risorse di sistema, con particolare enfasi sull'impiego delle CPU, e una pressione sempre maggiore sul contenimento dei costi.

Appendice A

La legge di Amdahl

7.1 Definizione di SpeedUp

In letteratura si indica con il termine *SpeedUp* il miglioramento di velocità ottenibile convertendo un algoritmo sequenziale nella sua versione parallelizzata: ciò presuppone che parti dell'algoritmo considerato possano essere eseguite in flussi indipendenti per poi riunire i risultati parziali nel momento in cui l'elaborazione raggiunga un punto di join.

Lo SpeedUp è quindi definito come segue

$$\mathrm{Su(N)} = \frac{T_1}{T_N} \qquad \textbf{EQ 27}$$

Dove:

- N è il numero di CPU disponibili
- T_1 è il tempo di esecuzione misurato utilizzando l'algoritmo in forma seriale
- T_N è il tempo di esecuzione dello stesso algoritmo in forma parallelizzata, utilizzando N CPU

Teoricamente, esiste un punto di massimo, detto SpeedUp ideale quando

$$\mathrm{Su(N)} = \mathrm{N} \qquad \textbf{EQ 28}$$

In questo caso, l'algoritmo con SpeedUp pari a N, cioè al numero di cores aggiunti, risulterebbe essere completamente parallelizzabile su una architettura con N CPU. Un algoritmo con SpeedUp ideale per un determinato valore N_1 può presentare uno SpeedUp inferiore con un valore di $N_1<N_2$

$$\begin{cases} \mathrm{Su(N_1)} = \mathrm{N_1} \\ \mathrm{Su(N_2)} < \mathrm{N_2} \end{cases} \quad \mathrm{N_1} < \mathrm{N_2} \qquad \textbf{EQ 29}$$

Sistemi che implementano questo tipo di algoritmi prendono il nome di algoritmi totalmente scalabili.
Un esempio di questo tipo di algoritmi è la FFT (*Fast Fourier Transform*) applicato a segnali con un numero di campioni riconducibili ad una potenza di due.

L'efficienza di un algoritmo parallelizzato è data dalla seguente espressione:

$$\mathrm{E(N)} = \frac{Su(N)}{N} = \frac{T_1}{NT_N} \qquad \textbf{EQ 30}$$

E(N) è un parametro compreso nell'intervallo [0,1] e quantifica l'impatto di overhead dovuto alla parallelizzazione dell'algoritmo.

Esso risulterà nullo (efficienza massima) per E(N) = 1, e massimo (efficienza nulla) per E(N) = 0; in quest'ultimo caso il tempo utile per l'esecuzione dell'algoritmo diventa trascurabile rispetto all'overhead di parallelizzazione.
In generale l'andamento di E(N) per algoritmi non intrinsecamente parallelizzabili è pari a:

$$\mathrm{E(N)} = \frac{1}{\log\,(\mathrm{N})} = \frac{T_1}{NT_N} \qquad \textbf{EQ 31}$$

7.2 La legge di Amdahl

La legge di Amdahl viene formulata più di 50 anni fa con lo scopo di elaborare una equazione che esprimesse l'incremento prestazionale ottenibile a fronte di una migrazione da un sistema mono processore ad uno equipaggiato da più di una CPU, a parità di software eseguito.

La stessa legge, però, può trovare impiego anche in altri ambiti dove sia possibile esprimere l'aumento prestazionale di una qualsiasi risorsa di sistema, non necessariamente legata alle capacità computazionali, e il suo impatto sul tempo di esecuzione di un processo che utilizzi quella risorsa.

Se si definisce T come il tempo di esecuzione di un processo sul sistema di riferimento, questa quantità T può essere scomposta in una somma di due quantità distinte: un tempo T_r durante il quale il processo utilizza la risorsa *r* considerata, e un tempo T_{or} durante il quale viene utilizzata u' altra risorsa di sistema.

$$T = T_r + T_{or} \qquad \textbf{EQ 32}$$

Se la risorsa *r* utilizzata per il tempo *Tr* viene sostituita da una risorsa che espleta la stessa funzionalità, ma più veloce, il tempo di esecuzione relativo ad essa si riduce:

$$\hat{T}_r < T_r \qquad \textbf{EQ 33}$$

Di conseguenza, sostituendo il nuovo valore $\hat{T}_r$ in EQ 32, il nuovo tempo di esecuzione del processo diventa:

$$\hat{T} = \hat{T}_r + T_{or} < T \qquad \textbf{EQ 34}$$

Se si procede a definire l'accelerazione S (*SpeedUp*) del sistema o di una sua parte a fronte dell'introduzione della nuova risorsa come il rapporto tra T e $\hat{T}$, la sua espressione diventa:

$$S = \frac{T}{\hat{T}} \qquad \textbf{EQ 35}$$

L'accelerazione della singola risorsa S_r, di conseguenza, prende la forma di:

$$S_r = \frac{T_r}{\hat{T}_r} \qquad \textbf{EQ 36}$$

Il processo utilizzerà la risorsa per una frazione *f* in [0,1] del tempo totale di esecuzione del processo. Si avrà quindi:

$$f = \frac{T_r}{T} \qquad \textbf{EQ 37}$$

Considerando l'espressione di *Tor* in EQ 32 e sostituendolo in EQ 33 si ottiene:

$$T - T_r = \hat{T} - \hat{T}_r \qquad \textbf{EQ 38}$$

Ma da EQ 37 si ottiene:

$$Tf = T_r \qquad \textbf{EQ 39}$$

Considerando EQ 35, EQ 36 ed EQ 39, l'EQ 38 diventa:

$$T - Tf = \frac{T}{S} - \frac{Tf}{S_r} \qquad \textbf{EQ 40}$$

Semplificando la variabile T ed esprimendo l'EQ 40 in funzione di A, si ottiene l'espressione:

$$S = \frac{1}{1 - f(1 - \frac{1}{S_r})} \qquad \textbf{EQ 41}$$

Lo *SpeedUp* S è quindi limitato superiormente dalla frazione di tempo *f* spesa dal processo nell' utilizzo della risorsa *r*.

Infatti:

$$\lim_{S_r \to \infty} S = \frac{1}{1 - f} \qquad \textbf{EQ 42}$$

Per meglio evidenziare l'impatto della percentuale di tempo *f* sulle prestazioni, l'EQ 41 può essere scritta nella forma:

$$S = \frac{1}{(1 - f) + \frac{f}{S_r}} \qquad \textbf{EQ 43}$$

Considerando come risorsa incrementale la CPU e l'incremento di accelerazione S legato all' aumento del numero N di cores disponibili per l'esecuzione dei processi, il valore di S dipenderà dal valore di N e l'EQ 43 prenderà la forma:

$$S(N) = \frac{1}{(1-f) + \frac{f}{N}} \qquad \textbf{EQ 44}$$

Nel lavoro proposto, la parte di tempo *f* che il processo spende nell' utilizzo della risorsa *r (*nel caso specifico la CPU) è sostituita dalla somma dei rapporti pesati tra *unvoluntary context switch* e *voluntary context switch* per ogni thread eseguito sul sistema. Essa prende il nome di *System Overbooking Ratio* (SOr).

Di fatto l'insieme dei processi e thread eseguiti sul sistema è equiparata ad un singolo processo "parallelizzato", con ogni sua sotto-componente caratterizzata da uno specifico livello di "*I/O-Boundness*" e "*CPU-Boundness*".

In Figure 49 è mostrato l'andamento di *S(N)* al variare di N con passo 1 e di *f* con passo 0.1.

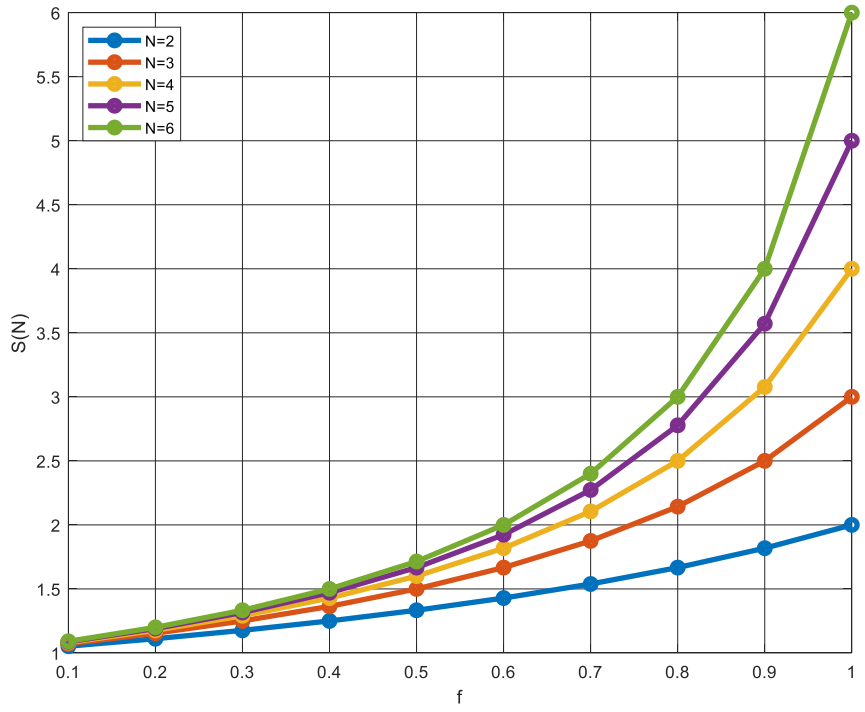

Figure 49 - Andamento di S(N) al variare di f

Indice delle figure

Bibliografia

[1] A. Silberschatz, P.B. Galvin, G. Gagne (2014), *Sistemi Operativi, Concetti ed Esempi*, Nona Edizione, Pearson

[2] M.H. DeGroot (1975), *Probability And Statistics*, Prima Edizione, Adison-Wesley

[3] A. S. Tanenbaum (2006), *Architettura dei calcolatori, Un Approccio Strutturale*, Quinta Edizione, Prentice Hall

[4] W. Stallings (2004), *Architettura e Organizzazione dei Calcolatori, Progetto e Prestazioni*, Sesta Edizione, Addison Wesley

[5] A. Murli (2006), *Lezioni di Calcolo Parallelo*, Prima Edizione, Liguori

[6] A. D. Pimentel, S. Vassiliadis (2004), *Computer Systems: Architectures, Modeling and Simulation*, Quarta Edizione, Springer

[7] P. Cornes (1997), *Linux A-Z*, Seconda Edizione, Prentice Hall

[8] Congiu S. (2007), *Architettura degli elaboratori, Organizzazione dell'hardware programmazione in linguaggio assembly*, Quinta Edizione, Pàtron Editore

[9] K. Popovici, F. Rousseau, A. A. Jerraya e M. Wolf (2010), *Embedded Software Design and Programming of Multiprocessor System-on-Chip*, Prima Edizione, Springer