ANALISI DI UN'APPLICAZIONE GRAFICA IN AMBIENTE ERIKA³

RELAZIONE TECNICA

Stefano Panozzo
Dipartimento di Matematica
Università di Padova, Italia I-35121
Email: stefano.panozzo.1@studenti.unipd.it

ABSTRACT

Questa relazione descrive un'applicazione *real-time* sviluppata con il supporto di ERIKA³, un *Hard Real Time Operating System (RTOS)* certificato OSEK/VDX in utilizzo in ambito automobilistico. L'applicazione consiste in diversi *task* che competono fra loro per raggiungere un obiettivo *target*. L'applicazione è testata in diverse configurazioni e condizioni di carico per vedere come queste influenzino l'esecuzione del sistema; in particolare utilizzo come metrica il numero di errori di attivazione dei *task*, equiparabili a *deadline-miss* quando un *task* supporta una sola attivazione e le *deadline* sono implicite.

1 Introduzione

I sistemi *embedded* sono di utilizzo comune nella vita di tutti i giorni: dai semplici sensori domotici nelle case ai più complessi sistemi che controllano un'automobile. Se però nei primi può non vedersi la necessità di garantire che il sistema funzioni sempre nel modo corretto e leggeri ritardi nell'esecuzione dei compiti sono ammissibili, in un sistema complesso che svolge compiti critici che coinvolgono direttamente la persona risulta necessaria la comprensione di ciò che accade oltre a ciò che ci è immediatamente visibile.

Al giorno d'oggi le automobili si possono considerare l'oggetto nella vita di tutti i giorni più tecnologicamente avanzato. Basti pensare all'enormità di sensori che sono presenti nei veicoli di ultima generazione: dai dispositivi per la sicurezza del passeggero, come l'ABS o il *lane assist*, ai sistema di intrattenimento di bordo. E se pensiamo al futuro, l'utilizzo di tecnologie sempre più avanzate non diminuirà, come ad esempio i sistemi a guida autonoma o la comunicazione tra veicoli o tra un veicolo e l'infrastruttura presente lungo la strada. Si riesce quindi a capire come sia necessario gestire tutti questi sistemi in modo tale da mantenere un certo ordine e garantire che sistemi a criticità massima, come quelli relativi alla sicurezza del passeggero, non siano disturbati da un malfunzionamento di sistemi secondari non critici, e che quindi sia necessario garantire che alcuni vincoli durante tutta l'esecuzione del sistema siano soddisfatti.

Per questo motivo, nel corso degli anni, sono nati degli standard per garantire un modello omogeneo di architettura software in ambito *automotive*, partendo dal più vecchio standard OSEK/VDX[1] alle nuove versioni di AUTOSAR[2].

Erika³[3] è un *Hard Real Time Operating System (RTOS)* che implementa le funzionalità richieste dagli standard sopracitati, fornendo delle primitive che semplificano lo sviluppo e offrono un livello di astrazione rispetto all'hardware sottostante.

Per lo svolgimento di questa analisi ho quindi scelto l'utilizzo di Erika³ su una *bare-board* STM32F429I-Discovery[4], un kit per lo sviluppo *low-cost* con un microcontrollore STM32F4 e un *core* ARM Cortex-M4.

[1] https://web.archive.org/web/20160307021413/http://osek-vdx.org/
[2] https://www.autosar.org/
[3] http://erika-enterprise.com/
[4] https://www.st.com/en/evaluation-tools/32f429idiscovery.html

2 ERIKA³

Come già detto, Erika³ supporta lo standard OSEK/VDX, il che comporta delle scelte progettuali ben definite.

2.1 Fixed-priority scheduling

In questo tipo di *scheduling*, ogni *task* ha una priorità fissata e statica, determinata *offline*. La priorità di un *task* deriva dai suoi parametri temporali, quindi non ha direttamente a che fare con la sua importanza relativa per l'integrità del sistema (criticità).

Un *task* rappresenta l'unità architetturale di lavoro da svolgere, ossia il compito che il sistema deve fare, che viene poi realizzato dal *job*, ovvero ciò che il sistema (o meglio lo *scheduler*) fa eseguire a *runtime*. Un *job*, di conseguenza, eredita la priorità del task che lo definisce.

Durante l'esecuzione il *job* a priorità più alta viene eseguito, mentre gli altri attendono finchè non ci sarà più nessun *job* a priorità maggiore nella coda dei pronti.

Per assegnare la priorità dei *task* si possono seguire due strategie:

- *Rate monotonic*: il *task* con periodo minore (il cui *job* viene attivato più frequentemente) ha priorità maggiore; il rate di un *task* è l'inverso del periodo.
- *Deadline monotonic*: il *task* con la *deadline* relativa più vicina ha priorità maggiore.

Quando la *deadline* di un *task* è implicita (i.e. *deadline* = periodo), le due strategie si equivalgono. Quando però la *deadline* non è implicita, l'algoritmo *deadline monotonic* offre prestazioni migliori, nel senso che può produrre, entro certi limiti, uno *schedule feasible* anche quando non è possibile con *rate monotonic*.

Fixed-priority scheduling supporta il prerilascio, ovvero se mentre un *job* esegue viene aggiunto alla coda dei pronti un altro *job* a priorità maggiore, il primo può essere interrotto e, dopo aver effettuato il *context switch*, il secondo viene fatto eseguire.

2.2 Shared/Private stack e Immediate Priority Ceiling

Una delle particolarità dei sistemi *embedded* è la carenza di memoria RAM. È fondamentale quindi attuare soluzioni che preservino il più possibile questo tipo di memoria.

Per fare ciò si può optare per uno *stack* condiviso fra due o più *task*, in modo da ottimizzare la memoria allocata staticamente a questi. Così facendo, la memoria utilizzata dai *task* non in esecuzione sarà libera ed utilizzabile dagli altri.

Guardando la Figura 1 e ragionando sul funzionamento dello *stack*, si capisce però che uno *stack* condiviso, quando lo *scheduling* permette il prerilascio, può portare problemi.

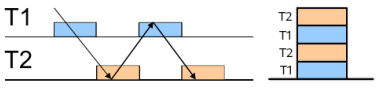

Figura 1: *Interleaved Execution* e stato dello *stack* dopo il secondo prerilascio di T1

Essendo l'ordinamento dello *stack* di tipo FIFO, nel caso di *stack* condiviso e *interleaved execution*, si avrebbe una frammentazione dello *stack*.

Per capire quando può avvenire una situazione del genere, bisogna ragionare su quando un *task* può essere sospeso per farne eseguire un altro. Oltre all'arrivo di un *task* a priorità maggiore, un *task* può essere sospeso se, ad esempio, non può accedere ad una risorsa necessaria per procedere perchè in possesso di un altro *task*. Questo problema può portare anche a *priority inversion*, dove un *task* a priorità più alta è bloccato da un *task* a priorità più bassa perchè in possesso di una risorsa necessaria per procedere. Questo problema è illustrato nella Figura 2, in cui T1 ha una *deadline miss*, e in cui l'ordine delle priorità è T1>T2>T3.

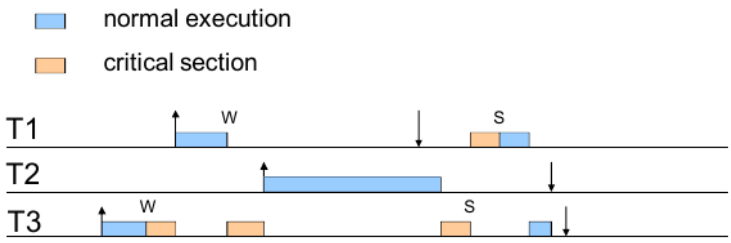

Figura 2: *Priority Inversion*

Per evitare questo problema e mantenere il prerilascio, si può adottare un algoritmo che permetta, una volta ottenuto l'utilizzo della risorsa, di eseguire senza essere interrotti da un altro *job* che necessita della stessa. Nello standard OSEK/VDX si è scelto di optare per *Immediate Priority Ceiling Protocol*, che permette di inibire il prerilascio nelle sezioni critiche di *job* in cui si utilizza una risorsa condivisa. Questo grazie al fatto che, quando un *job* ottiene una risorsa libera, aumenta la sua priorità al *priority ceiling* di questa, definita come la priorità del *task* a priorità maggiore che la utilizza. Così facendo, il *job* in esecuzione non potrà mai essere interrotto da un altro *job* durante la sezione critica.

2.3 OIL File

Lo standard OSEK/VDX impone un approccio statico, ovvero ogni specifica deve essere nota *offline*, prima che il sistema cominci l'esecuzione. Il file OIL è dove queste specifiche, rappresentanti la configurazione del *kernel*, sono definite.

Gli oggetti le cui proprietà sono definite nel file OIL sono i seguenti:

- CPU: la CPU su cui l'applicazione esegue;
- OS: specifiche del sistema operativo che esegue;
- ISR: *Interrupt Service Routine* supportate dal sistema operativo;
- RESOURCE: le risorse che possono essere usate da un *task*;
- TASK: la configurazione dei *task* gestiti dal sistema operativo;
- COUNTER: rappresenta il *tick* che attiva gli allarmi. Può essere hardware o software;
- EVENT: gli eventi definiti per un *task*;
- ALARM: gli allarmi che vengono attivati da un *counter*; quando azionati, possono attivare *task*, invocare funzioni di *callback* o cambiare lo stato di un evento;
- MESSAGE: il messaggio COM che fornisce la comunicazione locale o di rete;
- COM: il sottosistema di comunicazione inter- and intra- ECU;
- NM: il sottosistema per il *network management*.

Esiste inoltre una suddivisione in categorie di alcuni servizi che il sistema offre:

- TASK - I *task* possono essere di quattro categorie differenti, definite *Conformance Classes*:
  1. BCC1: il tipo più semplice di *task* (*Basic task*), consente una sola attivazione (i.e. se un *job* viene attivato mentre il *task* non è nello stato *suspended*, questa nuova attivazione viene persa) e un *task* per priorità;
  2. BCC2: *Basic task* ma con la possibilità di attivazioni multiple (le nuove attivazioni rimangono pendenti finchè il vecchio *job* non termina) e più di un *task* per priorità;
  3. ECC1: analogo al BCC1, ma la classe di *Extended task* consente la gestione degli eventi;
  4. ECC2: *Extended task* ma con la possibilità di attendere eventi multipli e più di un *task* per priorità.

  La Tabella 1 riporta le principali differenze fra le diverse *Conformance Classes*.

  Come si può intuire, aggiungendo funzionalità ad un tipo di *task*, la complessità a livello di gestione da parte del sistema operativo aumenta. Ad esempio, per supportare attivazioni di *task* con la stessa priorità, deve essere definita una struttura dati che mantenga l'ordine di attivazione, che poi verrà gestita dallo *scheduler* secondo l'ordinamento FIFO.

	BCC1	BCC2	ECC1	ECC2
Multiple requesting of task activation	no	yes	BT[3]: no ET: no	BT: yes ET: no
Number of tasks which are not in the *suspended* state	8		16 (any combination of BT/ET)	
More than one task per priority	no	yes	no (both BT/ET)	yes (both BT/ET)
Number of events per task	—		8	
Number of task priorities	8		16	
Resources	RES_SCHEDULER	8 (including RES_SCHEDULER)		
Internal resources	2			
Alarm	1			
Application Mode	1			

Tabella 1: *Conformance Classes - Fonte: OSEK 2.2.3 specification*

La maggiore complessità si può intuire anche valutando i possibili stati in cui *Basic task* ed *Extended task* possono essere, come riportato nella Figura 3.

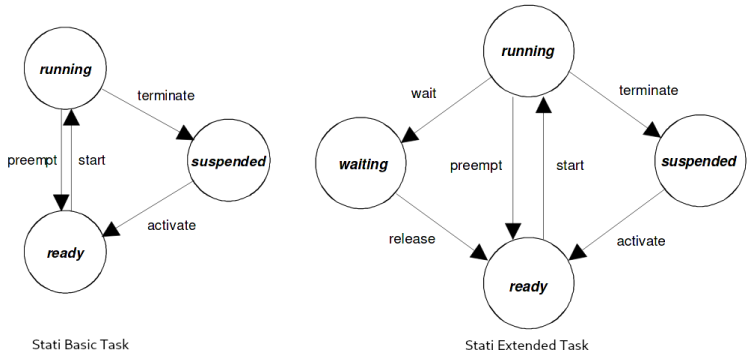

Figura 3: *Possibili stati nelle diverse classi - Fonte: OSEK 2.2.3 specification*

- Interrupt - sono suddivisi in due categorie differenti:

  1. ISR1: interrupt che non effettuano chiamate a primitive di sistema e, una volta terminati, l'esecuzione prosegue esattamente da dove era stata interrotta;
  2. ISR2: interrupt più complessi dei precedenti, possono effettuare chiamate a delle primitive di sistema e, proprio per questa capacità, al termine dell'esecuzione viene invocato lo *scheduler*.

  Gli ISR1 hanno priorità sempre maggiore degli ISR2, infatti i primi possono interrompere anche operazioni del *kernel*, mentre i secondi vengono eseguiti in seguito alla terminazione di queste.

Qualsiasi sia la priorità dei servizi offerti dall'OS, il compilatore assicura che ci sia un ordine ben preciso a *runtime* durante l'esecuzione di questi. Risulta quindi che gli interrupt avranno sempre la precedenza sui *task*, e gli ISR1 avranno a loro volta la precedenza sugli ISR2, come si può vedere dalla Figura 4 .

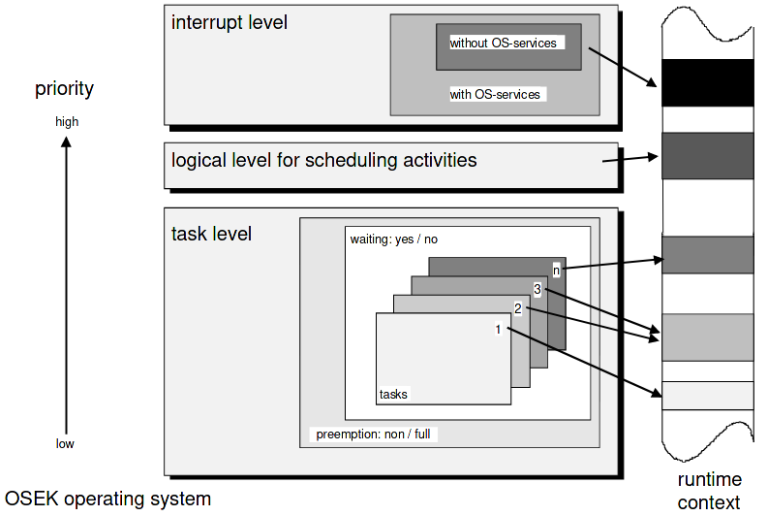

Figura 4: *Processing Levels - Fonte: OSEK 2.2.3 specification*

3 Applicazione

L'applicazione proposta utilizza quindi ERIKA³ e lo standard OSEK/VDX. Oltre a ciò, utilizza anche il *framework* TouchGFX[5], che aiuta lo sviluppo di interfacce grafiche complesse per sistemi hardware basati su microcontrollori STM32. Questo *framework* mantiene sincronizzata l'interfaccia grafica del display con il modello dei dati rappresentanti gli oggetti sullo schermo e comunica eventuali interazioni con essi attraverso il *touchscreen*.

L'applicazione di test consiste in un *task* che genera un punto target sullo schermo, indicato tramite un cerchio rosso, e tre *task* (o sei in base al test) con parametri di configurazione e implementazione differenti che lo devono raggiungere. È di interesse capire come l'applicazione e il sistema reagiscono all'aumentare delle interferenze e in differenti condizioni di carico. Mi attendo quindi un rallentamento della reattività del sistema e nello specifico, in base alle differenti configurazioni, una visibile differenza nello spostamento dei giocatori visualizzati a schermo che rappresentano i *task* del sistema.

Per avere una valutazione quantitativa e analizzare quello che accade a schermo, come metrica di confronto fra le varie configurazioni utilizzo il numero di *deadline miss* dei *task*, misurata con l'ausilio della primitiva *ErrorHook* in seguito definita. Tramite questa, infatti, è possibile sapere e gestire quando un nuovo *job* viene attivato, ad esempio da un allarme, mentre un vecchio *job* dello stesso *task* è ancora attivo (i.e. non nello stato *suspended*). Con la condizione di *deadline* implicite e *task* attivati solamente da allarmi periodici, questo significa che il vecchio *job* ha una *deadline miss*. In seguito chiamo questi eventi errori di attivazione. Tramite questa metrica, quindi, cercherò di valutare il comportamento del sistema nell'insieme e in seguito, più nello specifico, quali sono i *task* che subiscono più interferenze.

3.1 Configurazione del sistema

La *conformance class* dei *task* del sistema è di tipo ECC1, in quanto il *task* che gestisce l'interfaccia grafica non termina mai e rimane in attesa di eventi che notificano un nuovo *framebuffer* per aggiornare il display. Di seguito riporto la definizione del Task1, che si occupa della gestione dell'interfaccia grafica.

```
1  TASK Task1 {
2         PRIORITY = 3;                 /* Priorita del task */
3         ACTIVATION = 1;               /* Numero di attivazioni supportate */
4         SCHEDULE = FULL;              /* Task prerilasciabile */
5         AUTOSTART = TRUE;             /* Attivazione allo startup del sistema */
6         STACK = PRIVATE {             /* Stack privato e relativa dimensione */
7                 SIZE = 4096;
8         };
9         EVENT = FrameBufferEvent;     /* Supporto all'evento 'FrameBufferEvent' */
```

[5] https://support.touchgfx.com/docs/introduction/welcome

```
10        EVENT = VSyncEvent;          /* Supporto all'evento 'VSyncEvent' */
11        RESOURCE = HalResource;      /* Utilizzo della risorsa 'HalResource' */
12 };
```

Gli eventi dichiarati per questo *task* sono necessari per mantenere sincronizzato il display e l'*engine*, e per supportare questi è necessario quindi che il suo *stack* sia privato. Il numero di attivazioni supportate si riferisce a cosa accade quando viene attivato il *task* se questo è già in esecuzione: la configurazione scelta (una sola attivazione) implica che ogni attivazione pone il *task* nello stato *ready* solamente se quel *task* non è già attivo, altrimenti l'attivazione è persa.

Gli altri *task* presenti nel sistema sono:

- TaskBackground: ha priorità minima e non termina mai, per queste caratteristiche esegue solamente quando nessun altro *task* deve eseguire. È utile per il calcolo di utilizzo della CPU;
- Task2, Task3, Task4: rappresentano i *player* dello studio;
- Task5, Task6, Task7: rappresentano gli eventuali *player* aggiuntivi (non presenti in ogni condizione di test);
- Task0: rappresenta il target che i *player* devono raggiungere.

Tutti questi *task* hanno una configurazione di base simile: *stack* condiviso e un'unica attivazione (come per il Task1), ma cambiano nelle varie prove di questo studio in base al supporto o meno al prerilascio ed eventuali *offset* dall'inizializzazione del sistema.

Oltre alla configurazione dei *task*, serve definire i COUNTER necessari, in questo caso solamente uno che mantenga il *timing* del sistema:

```
1  COUNTER SystemTimer {
2          MINCYCLE = 1;
3          MAXALLOWEDVALUE = 65535;
4          TICKSPERBASE = 1;
5          TYPE = HARDWARE {
6                  SYSTEM_TIMER = TRUE;
7                  PRIORITY     = 9;
8                  DEVICE       = "SYSTICK";
9          };
10         SECONDSPERTICK = 0.001;
11  };
```

Come si può notare, ogni tick di sistema corrisponderà ad un tick hardware ogni millisecondo. Sono definiti anche gli allarmi che definiscono le *release* dei vari *task*:

```
1  ALARM AlarmTask2 {
2          COUNTER = SystemTimer;
3          ACTION = ACTIVATETASK { TASK = Task2; };
4          AUTOSTART = TRUE { ALARMTIME = 20; CYCLETIME = 10; };
5  };
```

Un esempio è l'ALARM per il Task2, attivato dal SystemTimer, in questo esempio a partire da 20ms rispetto all'inizializzazione del sistema e, in seguito, riattivato ogni 10ms.

Oltre a questi sono definiti anche tutti gli ISR1 e ISR2 necessari per la sincronizzazione e comunicazione del microcontrollore con il display, dell'invio dei dati dalla *board* all'esterno tramite seriale e dell'utilizzo del pulsante fisico presente sulla *board*.

Per trasmettere i dati di *log* dalla *board* all'esterno nel modo più efficiente possibile, ho deciso di implementare una coda circolare che trasmette tramite il modulo DMA (*Direct Memory Access*). Così facendo, l'utilizzo della CPU è limitato al massimo, in quanto è necessario il suo intervento solamente al termine della trasmissione del flusso dati, all'attivazione del relarivo *interrupt*. In questo modo riesco a sfruttare al meglio l'intera banda disponibile sulla *board* per la trasmissione UART senza interferire significativamente sull'esecuzione del sistema.

L'inserimento dei dati nella coda circolare è effettuata direttamente dai *task* o tramite i vari *hook* di sistema, definiti nello standard OSEK/VDX e implementati da ERIKA³. Quelli utilizzati in questo progetto sono:

- StartupHook: viene attivato al termine dell'inizializzazione dell'OS, prima che esegua lo *scheduler*;

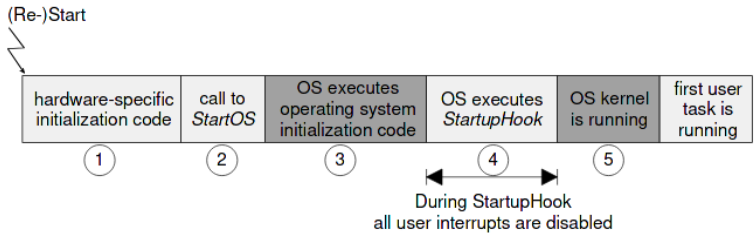

Figura 5: *StartupHook - Fonte: OSEK 2.2.3 specification*

- PreTaskHook e PostTaskHook: vengono eseguiti rispettivamente prima e dopo l'esecuzione di un *task*. Entrambi vengono eseguiti subito dopo o prima che un *task* entri o esca dallo stato *running*, quindi, quando invocati, è possibile ottenere il *task* a cui si riferiscono. Da precisare che questi, facendo riferimento allo stato *running* di un *task*, vengono invocati anche quando un *task* viene prerilasciato o ritorna ad eseguire, e non solo quando è attivato inizialmente o termina. Questo risulta molto utile per il *log* dei *task* TaskBackground e Task1 (GUI);

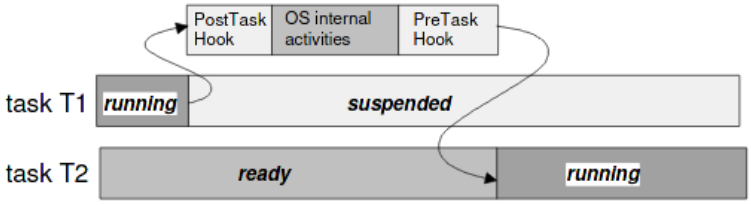

Figura 6: *PreTaskHook e PostTaskHook - Fonte: OSEK 2.2.3 specification*

- ErrorHook: viene eseguito ogni qualvolta ci siano errori del sistema operativo, sia temporanei che fatali. È possibile gestire differenti tipi di errore, in base allo '*StatusType error*' che l'ha generato. Nello studio in questione, definito che ogni *task* supporta una sola attivazione e ogni *task* ha *deadline* implicita uguale al periodo, è di interesse sapere quando l'ErrorHook riporta un errore di tipo 'E_OS_LIMIT', che viene invocato ogni qual volta viene eseguita un'attivazione di un *task* già in esecuzione. In questo modo si può determinare se un *task* non ha terminato entro la *deadline*.

Ho quindi utilizzato PreTaskHook, PostTaskHook ed ErrorHook per il *log* del sistema per avere un *timestamp* dell'esecuzione di alcuni *task* e per identificare eventuali *deadline-miss*.

3.2 Test

Ho testato il sistema in differenti condizioni, suddivisibili in base al loro scopo in due categorie:

1. Test del sistema senza carico, ovvero mantenendo la computazione dei *task* ridotta ma analizzando come, diminuendo il *release time* fra un *job* e il successivo, il sistema reagisce;
2. Test del sistema all'aumentare delle interferenze, che possono essere causate da altri *task* o dalla pressione del pulsante esterno e quindi dall'azionamento del relativo ISR2 che attiva il Task0. Si mantiene dunque costante il *timing* di attivazione dei *task* ma si valuta come al variare del carico dei *task* o delle pressioni esterne il sistema viene perturbato.

Per ogni test ho memorizzato, trasmettendo tramite DMA, i dati in un file di log esterno, comprendente un id incrementale della entry, il *task* che ha eseguito o l'eventuale errore relativo ad un *task*, il *timestamp* del tempo di esecuzione del *task* o il *timestamp* dell'errore e l'eventuale carico % della CPU.

D'ora in poi, chiamerò *player* i *task* che concorrono per raggiungere le coordinate target dello schermo.

3.2.1 Test del sistema senza carico

Durante questo scenario ho testato il sistema per 10 secondi e ho mantenuto il carico dei *player* molto esiguo. Questi infatti, avendo a disposizione le coordinate X e Y del target, devono calcolare rispetto alla loro posizione il movimento

da eseguire (il *player* si può muovere di un solo pixel per direzione alla volta), quindi inviare la nuova posizione al modello dei dati poi elaborato dal *framework* TouchGFX.

Ho effettuato i test sia con 3 *player* - e quindi 6 *task* totali - sia con 6 *player* - e quindi 9 *task* totali.

Come prima cosa ho testato come reagisce il sistema senza nessun *offset* sui *player*, ovvero quando il *release time* iniziale è per tutti uguale a 0. Come si può notare dalla tabella e dalla Figura 7, la differenza nel numero di errori di attivazione fra il sistema con 3 *player* e quello con 6, comincia ad essere rilevante quando il periodo dei *player* risulta uguale a 2ms. Questo era prevedibile, in quanto essendo il carico sul singolo *task* esiguo, con *release event* ben distanziati fra loro, tutti i *player* hanno il tempo necessario per eseguire. Quando invece il periodo, e di conseguenza la *laxity*, ovvero quanto un *job* può ancora rimanere in attesa prima di dover eseguire per completare entro la *deadline*, viene ridotto al minimo, il maggior numero di *player* può causare maggior interferenza che conclude in un maggior numero di errori di attivazione e di conseguenza di *deadline miss*. È possibile notare come il maggior numero di *player* influisca di più anche sul carico della CPU man mano che la frequenza di attivazione aumenta.

	SENZA OFFSET - 6 PLAYER		SENZA OFFSET - 3 PLAYER	
Periodo [ms]	Deadline miss	% CPU	Deadline miss	% CPU
1	2263	32	1302	25
2	265	23	114	21
5	6	18	3	17
10	0	16	0	15

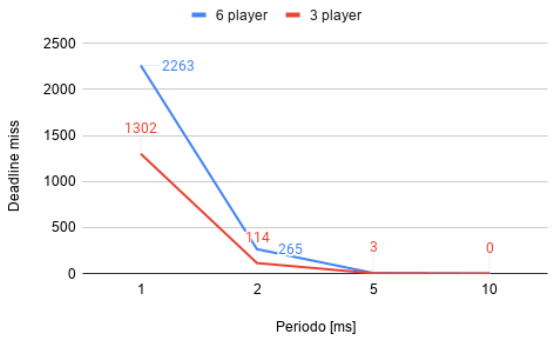

Figura 7: Confronto *deadline miss* dei *task* e % CPU con 3 e 6 *player* senza *offset*

È inoltre testato se l'aggiunta di un *offset* arbitrario possa diminuire il numero di errori e quindi di *deadline miss*; in questo caso i primi release event dei *player* sono stati posti a 20ms (per i *player* 1 e 4), 23ms (per i *player* 2 e 5) e 27ms (per i *player* 3 e 6). Essendo i *task* periodici, l'*offset* si è poi mantenuto nel tempo anche nelle successive attivazioni.

Come si può vedere nella Figura 8, un *offset* sull'attivazione dei *task*, in modo che i *release event* avvengano in momenti sufficientemente distanziati, migliora il numero di errori di attivazione. Con periodo uguale a 1ms mi sarei aspettato un minor miglioramento delle prestazioni, considerando che l'utilità dell'*offset* è nulla se i *release event* dei *player* si sovrappongono perchè il periodo è minore o uguale alla differenza fra l'*offset* di due *task*.

	CON OFFSET - 6 PLAYER		CON OFFSET - 3 PLAYER	
Periodo [ms]	Deadline miss	% CPU	Deadline miss	% CPU
1	2096	32	1266	25
2	199	24	83	21
5	2	18	1	17
10	0	16	0	15

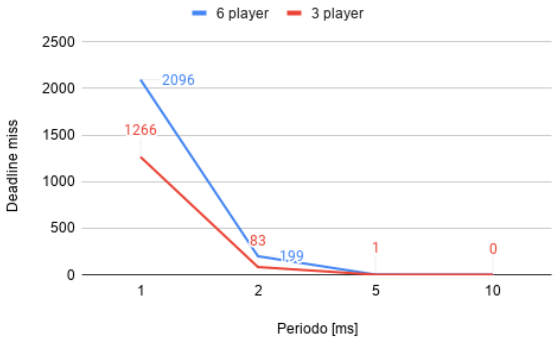

Figura 8: Confronto *deadline miss* dei *task* e % CPU con 3 e 6 *player* con *offset* arbitrario

3.2.2 Test del sistema all'aumentare del carico

Durante questo test ho mantenuto costante a 10 ms il periodo di attivazione dei *task* e ho definito solamente 3 *player*, utilizzando gli *offset* scelti nel precedente test. Questo perchè così facendo parto da una base in cui, senza carico, non ho ottenuto nessun errore di attivazione.

Ho invece aumentato il carico, in particolare sul *player* 3, e testato se eventuali pressioni del pulsante fisico presente sulla *board*, che attivano il Task0 (target), influiscono sul numero di errori di attivazione.

L'aumento del carico sul *player* 3 è determinato dal semplice codice qui di seguito:

```
1  for(int i = 0; i < load; ++i)
2          for(int z = 0; z < load; ++z)
3                  sum += i+z;
```

in cui la variabile 'load' rappresenta il carico sul *player* 3 che viene aumentato nei vari test. L'iterazione non viene ottimizzata dal compilatore, essendo la variabile 'sum' definita come *'volatile'*.

Ho testato il sistema con prerilascio attivato per tutti i *task*, con prerilascio disattivato per i *player* e infine con prerilascio ma con il *player* 3 con una priorità maggiore di quella degli altri *player*. Negli altri due test, invece, ho assegnato al *player* 3 una priorità minore rispetto agli altri.

È importante notare come il *player* 3, già con carico uguale a 300, impiega 8/9 ms per completare l'esecuzione, quindi ogni carico aggiuntivo rende il *task* non *feasible*. È di interesse quindi valutare come la presenza di un *task* non *feasible* influisce sull'intero sistema in esecuzione e se qualche configurazione aiuti a mitigare le *deadline miss* totali.

	CON PREEMPTION		SENZA PREEMPTION		PLAYER3 PRIORITÀ ALTA	
Carico Player3	Deadline miss	% CPU	Deadline miss	% CPU	Deadline miss	% CPU
200	0	51	0	51	0	51
300	27	94	1	94	27	94
400	467	87	1401	86	1308	84
500	622	90	1562	86	1563	87
600	702	95	1875	90	2014	92
700	771	90	2244	95	2235	97
800	798	95	2285	97	2239	93
900	832	93	2354	96	2351	94
1000	848	94	2414	96	2538	99
1100	859	95	2409	95	2543	100

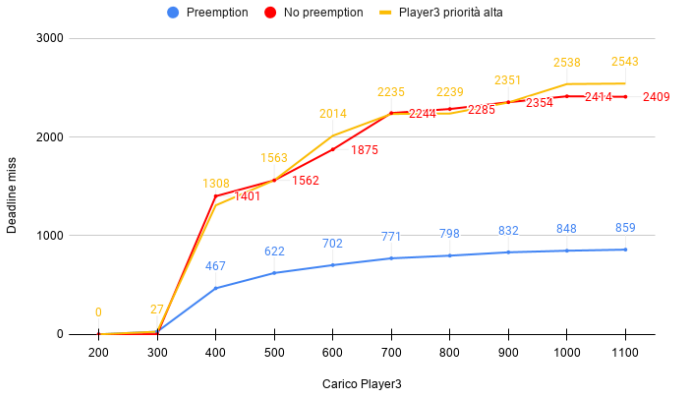

Figura 9: Confronto *deadline miss* dei *task* e % CPU con *preemption*, senza *preemption* e con *player* 3 con priorità alta, con *offset* arbitrario

Come si può notare dai risultati ottenuti in Figura9, la presenza di prerilascio diminuisce di molto il numero di errori di attivazione. La differenza fra l'esecuzione senza prerilascio e con prerilascio ma con il *player* 3 a priorità maggiore degli altri due *player* è esigua. Questo era prevedibile, essendo il *player* 3 il *task* bloccante e avendo un tempo d'esecuzione molto maggiore degli altri *task*.

Si nota come il carico % sulla CPU non mantiene un andamento lineare al crescere del carico sul *player* 3 quando lo *schedule* non è *feasible* (carico > 300). Ciò può essere dovuto al fatto che la nuova attivazione di un *task* mentre un suo vecchio *job* è ancora in esecuzione causa la perdita di quella attivazione, e dunque quel task una volta terminato rimarrà in attesa di un nuovo allarme senza eseguire. È da notare, comunque, che in alcuni test con un'altra applicazione, il carico ha raggiunto un valore di picco > 100%; ciò potrebbe indicare un errore nella libreria STM utilizzata per ottenere il valore, ma ciò non è stato approfondito.

Per completezza, ho testato il sistema anche senza utilizzare alcun *offset*.

	CON PREEMPTION		SENZA PREEMPTION		PLAYER3 PRIORITÀ ALTA	
Carico Player3	Deadline miss	% CPU	Deadline miss	% CPU	Deadline miss	% CPU
200	0	51	0	51	0	51
300	27	94	0	94	99	94
400	467	87	467	86	1401	84
500	622	89	1244	86	1866	87
600	702	94	1638	90	2106	91
700	771	90	1860	96	2232	98
800	798	95	2319	99	2412	92
900	829	92	2204	96	2462	95
1000	849	95	2350	94	2538	99
1100	861	95	2391	95	2544	100

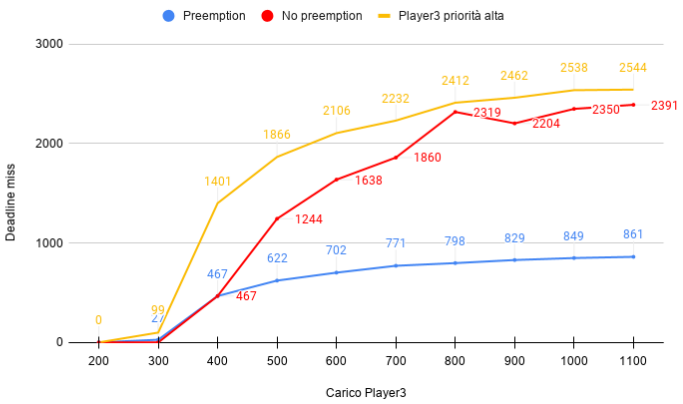

Figura 10: Confronto *deadline miss* dei *task* e % CPU con *preemption*, senza *preemption* e con *player* 3 con priorità alta, senza *offset*

La differenza più grande si nota durante l'esecuzione del sistema, visualizzando il comportamento dei *player* sullo schermo.

Con *preemption* attiva, il risultato a schermo è il medesimo sia con *offset* che senza: i *player* 1 e 2 si muovono alla stessa velocità mentre il *player* 3 risulta più lento.

Con *preemption* disattivata, l'utilizzo dell'*offset* mitiga l'effetto delle interferenze del *player*3 sugli altri *task*, in particolare sul *player*2, che, con alcuni carichi, risulta visibilmente più veloce.

Con il *player* 3 a priorità più alta, utilizzando l'*offset* si nota lo stesso effetto precedente, mentre senza *offset* il *player* 3 risulta visibilmente più veloce, a conferma di un maggior numero di errori di attivazione per gli altri due *player*.

La pressione del pulsante fisico non risulta causare errori di attivazione aggiuntivi, nemmeno quando l'input è stato attivato assiduamente.

Per trovare una corrispondenza fra ciò che si nota a schermo e quello che accade nel sistema sottostante, ho cercato di analizzare quali *task* subiscono maggiori interferenze, suddividendo le *deadline miss* per ogni *task*. Per fare ciò è necessario capire a quale *task* si riferisce l'errore riportato nell'*ErrorHook*, ovvero quale allarme ha cercato di attivare il *task* interessato. ERIKA³ fornisce alcune API che consentono di interrogare il sistema quando viene invocato l'*ErrorHook*, e in particolare la funzione 'OSErrorGetServiceId()' riporta quale servizio ha generato l'errore; in seguito, sono fornite alcune funzionalità per ottenere maggiori informazioni su quel servizio, come l'identificativo dell'allarme, del *task* o dell'*interrupt* a cui fa riferimento, e quindi la possibilità di capire chi ha causato l'errore. Purtroppo ERIKA³ non supporta alcuna API per la gestione degli errori derivanti da attivazioni periodiche di *task*. Dopo aver escluso questa opzione, ho quindi cercato di agire direttamente a livello del *kernel*, ma purtroppo oltre ad ottenere ciò che cercavo tramite un *debugger* controllando a *runtime* lo stato di alcune variabili di sistema, non sono riuscito ad automatizzare il lavoro e avere una visione per tutta la durata del test.

4 Conclusioni

Durante questo studio si è visto come l'utilizzo di *offset* e di *preemption* possono aiutare a ridurre gli errori di attivazione, e quindi di *deadline miss*, quando le interferenze fra *task* e il carico sono contenuti e, nella maggior parte dei casi, anche quando lo *schedule* non è *feasible*. Nel test del sistema senza carico, infatti, noto una sostanziale differenza fra l'esecuzione del sistema applicando un *offset* iniziale, mentre nel test all'aumentare del carico noto come la presenza del prerilascio mitighi di molto le *deadline miss*.

Lo studio ha margini di ricerca aggiuntiva, come ad esempio l'identificazione dei *task* che hanno subito maggior interferenza durante l'esecuzione (approfondimento non riuscito in questo studio), ma anche tramite una misurazione precisa del *jitter* dei *task* e di come vari il tempo d'esecuzione di questi.