

MrsP

Capitolo esperimenti e valutazione

SEBASTIANO CATELLANI

Universita' degli Studi di Padova

July 26, 2014

Indice

1 Esperimenti e valutazioni	3
1.1 Ambiente di esecuzione	4
1.1.1 Generazione ed esecuzione degli esperimenti	5
1.2 Confronto tra protocolli	7
1.2.1 Esperimento	7
1.2.2 Configurazione	8
1.2.3 Obiettivo	8
1.2.4 Risultati	10
1.2.5 Considerazioni	14
1.3 Calcolo degli overhead	17
1.3.1 Esperimento	17
1.3.2 Configurazione	17
1.3.3 Obiettivo	18
1.3.4 Risultati	19
1.3.5 Considerazioni	25
1.4 Confronto in assenza di risorsa	26
1.4.1 Esperimento	26
1.4.2 Configurazione	26
1.4.3 Obiettivo	28
1.4.4 Risultati	28
1.4.5 Considerazioni	30
Appendices	32
A LITMUS^{RT}	32
B Librerie user-space	34
C TRACE() e Feather-Trace	36
C.1 TRACE()	37
C.2 Feather-Trace	37
D experiment-scripts	38
Bibliography	40

1 Esperimenti e valutazioni

Gli esperimenti eseguiti hanno lo scopo di valutare l'implementazione proposta di *MrsP* da diversi punti di vista.

In un primo insieme di esperimenti il protocollo viene messo a confronto con altri due protocolli che, come *MrsP*, sono sviluppati su sistemi partizionati con condivisione di risorse globali: il primo è basato su un approccio *simple ceiling*, mentre il secondo utilizza inibizione di prerilascio. In questo esperimento si considerano le prestazioni dei protocolli nell'esecuzione di taskset creati su misura per confrontare i *response time* dei vari task in specifiche circostanze. Infine i risultati delle elaborazioni vengono confrontati con i dati ottenuti dalle simulazioni in Burns et al. [4].

In seguito ~~si discute~~ avviene una valutazione ~~costi~~ delle performance dell'implementazione: una serie di campionamenti permettono di verificare il costo aggiunto da *MrsP* alle primitive dello scheduler, in particolare gli overhead per integrare il protocollo a P-FP (*Partitioned Fixed Priority*).

Infine risulta interessante il funzionamento dello scheduler in assenza di risorse, questo in quanto la presenza di risorse globali condivise in un sistema partizionato è un caso particolare di esecuzione, quindi un buon funzionamento in sua assenza risulterebbe positivo ai fini di una valutazione completa. A tal fine il sistema viene messo a confronto con P-FP, cioè il medesimo *scheduler* privo di integrazione ~~a~~ *MrsP*.

1.1 Ambiente di esecuzione

Gli esperimenti sono stati effettuati con supporto di una macchina dotata di piattaforma i7-2670QM, ~~la~~ architettura è stata lanciata da Intel nell'ottobre del 2011.

prova Sandy Bridge è l'architettura alla base del sistema, ~~esso~~ consiste in un quad-core con frequenza di clock pari a 2.2 GHz, 3.1 GHz in Turbo mode. Ogni core possiede due livelli di cache, L1 e L2 di dimensione rispettivamente pari a 64 KB e 256 KB, ed un terzo livello L3 condiviso tra i 4 core con dimensione di 6 MB. La memoria ~~chace~~ utilizza un metodo di gestione creato da Intel e chiamato "Smart ~~chace~~": permette di diminuire il rapporto globale di ~~chace miss~~, aumentando così l'efficienza del suo utilizzo. La tecnologia Simultaneous Multi-Threading permette di raddoppiare il numero di core, trasformando i 4 fisici in 8 logici, tramite l'esecuzione di due thread sul medesimo core. Questa opzione viene disabilitata in fase di test, allo stesso modo anche le funzioni di gestione della potenza sono state disattivate. Il ~~BUS~~ interno alla CPU ha una velocità pari a 100 MHz, mentre il ~~BUS~~ seriale che permette le comunicazioni tra processori e con il chipset raggiunge i 5 GT/s. I ~~BUS~~ per le connessioni tra processori e chipset utilizzano la tecnologia QuickPath Interconnect (QPI), la cui caratteristica principale consiste nel permettere comunicazioni "point-to-point" tra le varie componenti; questo è indubbiamente un vantaggio rispetto all'utilizzo del ~~BUS~~ come canale unico per tutte le comunicazioni, permettendo così trasferimenti simultanei. *qui* La piattaforma utilizzata supporta la Vanderpool Technology, una particolare tecnologia di virtualizzazione per piattaforme Intel che rende possibile l'esecuzione simultanea di più sistemi operativi ~~differenti~~ contemporaneamente.

LINUX

Gli esperimenti sono stati eseguiti con il supporto di macchina virtuale; l'infrastruttura di virtualizzazione è basata su Kernel-based Virtual Machine (KVM) che è specifica per i sistemi LINUX, mentre il software di emulazione QEMU permette di eseguire il sistema operativo come ospite della macchina fisica. L'immagine virtuale utilizzata è in formato compatibile con QEMU ed esegue la bzImage generata tramite la compilazione del kernel LITMUS^{RT}.

Quale?

Il comando di lancio della VM è il seguente:

```
qemu-system-x86_64 -enable-kvm -smp 4 -m 512  
-boot c -nographic -net nic -net user,hostfwd=tcp::10022-:22  
-kernel bzImage -append "console=ttyS0,115200 root=/dev/hda1"  
-hda ubuntu.backing.qcow2.img
```

Tramite i parametri specificati il sistema di virtualizzazione riconosce che viene lanciato un kernel a 64 bit utilizzando KVM; quest'ultimo supporta la tecnologia di virtualizzazione specifica di Intel denominata Vanderpool citata precedentemente: permette di dividere un sistema in macchine virtuali distinte nonostante condividano le stesse risorse di sistema; ne risultano quindi due macchine che operano in maniera totalmente indipendente (logiche), grazie all'appoggio di specifiche funzionalità hardware che consentono di ottimizzare tale condivisione. In particolare l'architettura hardware abbinata a questa configurazione permette un accesso diretto ai core fisici senza alcun livello di virtualizzazione intermedio.

Alla macchina virtuale vengono assegnati il kernel LINUX -hda, 4 core fisici e 512 MB di memoria.

In fase di sviluppo sono stati utilizzati alcuni strumenti *user-space* per interagire con LITMUS^{RT}:

- *liblitmus*, appendice B, è una libreria che permette la creazione ed il controllo di task set;
- *TRACE()* permette di ottenere informazioni dall'esecuzione, è il principale strumento per effettuare debugging.

I campionamenti sono effettuati tramite *Feather-Tracē*, il quale consiste in una serie di strumenti atti a calcolare gli overhead delle primitive e rilevare gli eventi di scheduling. Tramite i dati raccolti è stato possibile valutare l'implementazione e confrontare i vari protocolli.

Maggiori informazioni sui metodi di tracciamento e campionamento sono presenti nell'appendice C.

presentati

1.1.1 Generazione ed esecuzione degli esperimenti

Per la creazione dei taskset sono stati usati tre differenti approcci:

- generati manualmente per ottenere un determinato comportamento;
- tramite applicazione Java per taskset che richiedano l'utilizzo di risorse;
- *experiment-scripts*, una suite di script in Python per la creazione di taskset.

e

La libreria *experiment-scripts* definisce un formato di file con il quale è possibile avviare con un unico comando un intero taskset:

- `sched.py` consiste in una lista di task con relativi parametri (WCET, periodo, risorse, etc.);
- `params.py` contiene informazioni che specificano il plugin utilizzato ed informazioni riguardanti il taskset.

Una volta deciso lo scheduler da utilizzare si genera manualmente o tramite script il file che specifica il taskset. Maggiori informazioni riguardanti `experiment-scripts` in D.

*appendice
sono supportate*

1.2 Confronto tra protocolli e viene

Con protocolli *lock-based* l'accesso viene gestito, in caso di risorsa occupata, tramite sospensione oppure attesa attiva. Nel primo caso se la risorsa è occupata il task richiedente si sospende ed inserito in una coda in attesa del suo rilascio. Al contrario in presenza di un protocollo *spin-based* il richiedente effettua attesa attiva fino al momento di accesso alla risorsa. Tale argomento viene maggiormente discusso in Brandenburg et al. [1]. MrsP si basa su approccio spin-based, in quanto permette di limitare il tempo di blocco subito, ma risulta cruciale come ed in quali circostanze effettuare attesa attiva.

approfonditamente

In questo esperimento ci si sofferma su questo aspetto, cioè in che modo effettuare attesa attiva ed a quali condizioni, a seconda del valore di priorità scelto per proseguire l'attesa si ottengono comportamenti differenti che influenzano in modo diverso il sistema.

1.2.1 Esperimento TITOLO O NUMERO PROSESSIVO

Il seguente esperimento mette a confronto tre protocolli differenti costruiti su sistema partizionato con dispatching basato su priorità, mentre l'accesso alla risorsa globale viene gestito tramite accodamento FIFO. In tutti e tre i casi i protocolli sono basati su SRP: un job inizia ad eseguire solamente quando le risorse di cui necessita sono libere, innalza la propria priorità al momento della richiesta ed effettua attesa attiva fino a quando ne ottiene l'accesso esclusivo.

Il protocollo basato su *simple ceiling* prevede che il job innalzi la propria priorità al valore della priorità più alta tra tutti i task allocati nella stessa CPU che la richiedono, esattamente come MrsP. Il suo comportamento è identico a quello di MrsP, salvo il fatto che non vi è nessun meccanismo di migrazione.

analoga effetto

Il terzo protocollo prevede che al momento della richiesta venga inibito il prerilascio, questo in fase di implementazione è stato ottenuto innalzando la priorità al valore massimo, non permettendo a nessun job di causare prerilascio.

localmente alla CPU del richiedente

L'esperimento prevede di mettere a confronto i response time dei task che compongono il sistema, analizzando quali verificano maggiormente penalizzati dall'attesa attiva nei tre protocolli a variare di parametri come la

lunghezza della sezione critica o del WCET di determinati task.

1.2.2 Configurazione task (τ_1, τ_2, τ_3)

Il taskset prevede 3 job divisi su due CPU: sulla prima CPU viene allocato un task a priorità maggiore ed uno a priorità inferiore, quest'ultimo condivide la risorsa globale con un job ~~identico~~ allocato nella seconda CPU.

L'esecuzione è impostata in modo tale che il primo job ad essere rilasciato ed eseguire sia quello a priorità inferiore allocato nella prima CPU, poi il job della seconda CPU ed infine quello a priorità più alta. L'esecuzione voluta è rappresentata nella figura 1:

- punto 1, il job a priorità più bassa viene rilasciato ed ottiene la risorsa in quanto libera;
- punto 2, il secondo job a bassa priorità si accoda ed effettua attesa attiva nella seconda CPU;
- punto 3, Il job a priorità più alta tenta di eseguire causando prerilasciare il primo job; quest'ultimo passaggio viene gestito in modo differente dai tre protocolli.

del

1.2.3 Obiettivo

Variando il tempo di esecuzione del job a priorità più alta e la lunghezza della sezione critica della risorsa globale il comportamento atteso è che in ogni protocollo sia differente il job che soffre maggiormente tale cambiamento.

tale GADVANCE

Nei capitoli precedenti è stato chiarito come la condivisione della risorsa tra più CPU in un sistema partizionato vada ad aumentare il costo pagato da alcuni job. In MrsP influenzano solamente i job che la vogliono ottenere e coloro che subiscono blocco da essi, cioè non la richiedono ma un job a priorità più bassa la contende ad uno a priorità superiore alla propria tra quelli della medesima CPU.

Privando MrsP dei meccanismi per gestire i prerilasci si ottiene un comportamento simile a quello del protocollo basato su *simple ceiling*: i job a priorità maggiore non verranno influenzati, al tempo stesso aumenta il tempo di blocco subito dai job a priorità inferiore al ceiling ed i tempi di attesa dei job

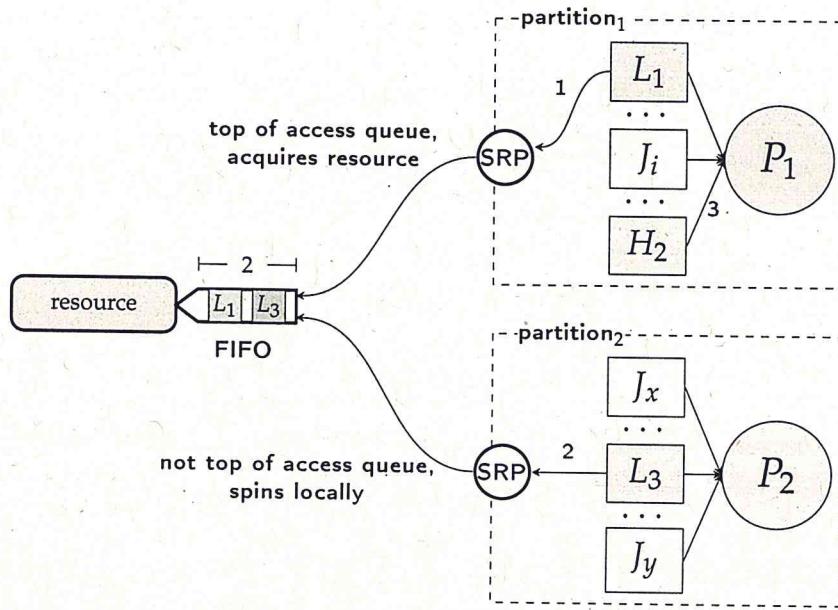


Figure 1: Configurazione del test tra protocolli.

accodati sulla risorsa allocati in altre CPU. Tale aumento di blocco ed attesa è determinato dal fatto che il proprietario della risorsa non può proseguire l'esecuzione della sezione critica in quanto prerilasciato. Il risultato è un sistema in cui l'interferenza subita dal lock holder si ripercuote anche sugli altri processori.

un ciardolo parla di
 Il comportamento atteso dal protocollo che inibisce il prerilascio è che il job a soffrire maggiormente della condivisione sia quello a priorità più alta. *localmente profondo*
 esso subisce l'esecuzione della sezione critica nonostante non necessiti della risorsa globale. Il risultato è quindi che soffrono blocco anche i job che non rispecchiano la condizioni fornite in precedenza, la loro esecuzione di conseguenza viene ritardata fino a che il job non ripristina la propria priorità al momento del rilascio della risorsa.

Un altro aspetto che si vuole studiare è come le migrazioni vadano ad influenzare le prestazioni di MrsP: un ultimo taskset è configurato in modo tale che le circostanze che forzano il job dalla prima CPU alla seconda a migrare si ripresentino anche in quest'ultima, obbligando ad una seconda migrazione nella terza CPU in cui eseguire la sezione critica.

Task	Partition	priority	Critical section	WCET
L_1	P_1	20	1	1
H_2	P_1	10	0	1
L_3	P_2	20	1	1

Table 1: Confronto tra protocolli: primo task set.

Task	MrsP	Ceiling	Non preemption
L_1	1.206.362	2.194.042	1.111.517
H_2	1.098.587	1.068.602	1.977.039
L_3	2.351.562	3.168.240	1.911.890

Table 2: Confronto tra protocolli: risultato primo task set, tempi espressi in nano secondi.

1.2.4 Risultati ~~Eperimento numero~~ X

Nella tabella 1 è rappresentato il primo task set. In esso i job hanno i medesimi tempi di esecuzione (pari ad un millisecondo). La durata della sezione critica è compresa nell'esecuzione del WCET; nel caso abbiano lo stesso valore si intende che il job effettui immediatamente la richiesta di accesso ed al momento del rilascio esegua le ultime istruzioni in modo tale da completare la propria esecuzione. Ne consegue che l'evento di rilascio della risorsa e quello di completamento non coincidono. Questo particolare è importante in quanto in alcuni casi nelle tempistiche riportate si considera il momento di rilascio della risorsa.

come una prima
azione
e completa

del task

Nella tabella 2 sono riportati i tempi di completamento di ogni job per i tre protocolli. I valori sottolineati indicano l'istante di rilascio della risorsa piuttosto che il completamento, questo in quanto il resto esecuzione subisce interferenza da parte del job a priorità superiore ed esula dai compiti dei protocolli di accesso a risorsa.

L'esecuzione del taskset è rappresentato graficamente in 2, 3 e 4. Analizzando i dati, si nota come in MrsP la migrazione renda minimo il tempo di attesa subito da L_3 e nullo il blocco subito da H_2 . I valori di esecuzione rappresentati subiscono degli overhead dati dal sistema, in particolare il costo della migrazione. Questo comportamento viene messo in risalto nella figura 2, le due linee rosse tratteggiate evidenziano l'overhead dato dal cambio di processore. Tale lasco di tempo consiste nel tempo che impiega il job a migrare, per cui L_3 continua ad effettuare attesa attiva e lo smaltimento della coda FIFO viene rallentato in quanto L_1 non sta progredendo

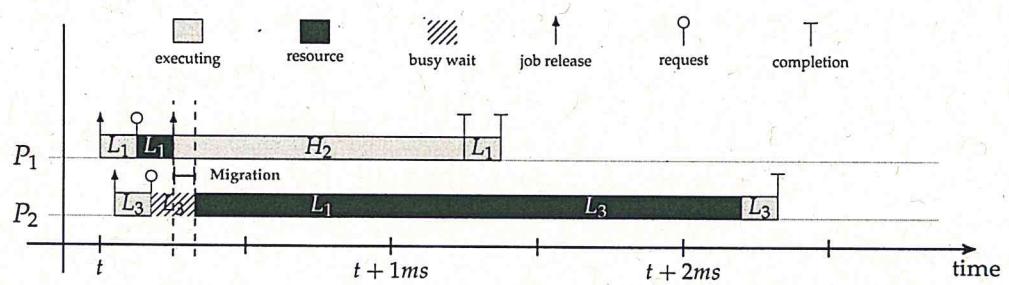


Figure 2: *MrsP*.

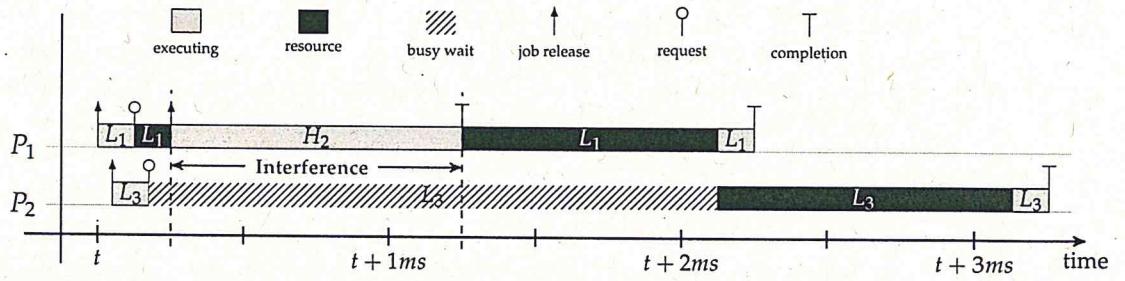


Figure 3: *Simple ceiling*.

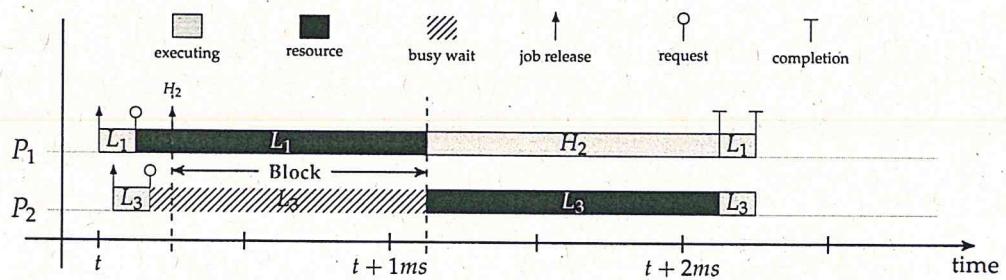


Figure 4: *non preemption*.

Task	Partition	priority	Critical section	WCET
L_1	P_1	20	3	3
H_2	P_1	10	0	1
L_3	P_2	20	3	3

Table 3: Confronto tra protocolli: aumento della sezione critica.

Task	Partition	priority	Critical section	WCET
L_1	P_1	20	1	1
H_2	P_1	10	0	3
L_3	P_2	20	1	1

Table 4: Confronto tra protocolli: aumento dell'interferenza.

nell'esecuzione della sezione critica.

Con l'approccio basato su simple ceiling il tempo di attesa di H_2 non dipende più solamente dalla lunghezza della coda della risorsa, ma anche dall'interferenza che il lock holder subisce.³ La figura 3 mostra come l'interferenza penalizzi ogni processore in cui vi sia un job in attesa della risorsa. Modificando il tempo di esecuzione di H_2 ci si aspetta che tale costo aumenti o diminuisca di conseguenza.

Al contrario dei casi precedenti, inibendo il prerilascio il job a soffrire maggiormente la condivisione della risorsa è H_2 in quanto non riesce ad eseguire nonostante non la richieda. L'inizio dell'esecuzione del job a priorità maggiore viene quindi ritardata; in figura 4 è evidenziato questo comportamento, pertanto all'aumentare della sezione critica H_2 subisce un maggiore blocco.

Nei taskset 3 e 4 sono state apportate modifiche rispettivamente alla lunghezza della sezione critica ed al tempo di esecuzione del job a priorità maggiore.

Nel primo caso il comportamento è quello che ci si aspetta:⁵ la figura 3 mostra come un aumento della sezione critica con MrsP dilunghi il tempo di attesa di L_3 , che viene utilizzato come nel caso precedente per far proseguire L_1 al momento del prerilascio, mentre H_2 resta inalterato. Al contrario, con simple ceiling sia L_1 che L_3 subiscono l'interferenza da parte di H_2 . Infine inibendo il prerilascio il job di L_3 subisce ulteriormente l'inversione di priorità.

Agendo sul WCET di H_2 (6) si dimostra come MrsP sia più performante al netto dei salvi i costi della migrazione, mentre con simple ceiling la maggiore interferenza subita da L_1 rende maggiore il tempo di attesa di L_3 . Inibendo il prerilascio il maggior tempo di esecuzione non influisce sui job che vogliono

Task	MrsP	Ceiling	Non preemption
L_1	<u>3.066.828</u>	4.242.092	<u>3.177.307</u>
H_2	1.035.721	1.141.324	3.956.506
L_3	6.099.752	7.209.873	6.024.691

Table 5: Confronto tra protocolli: aumento della sezione critica, tempi espressi in nano secondi.

Task	MrsP	Ceiling	Non preemption
L_1	<u>1.053.232</u>	4.215.599	<u>1.113.397</u>
H_2	3.018.344	3.071.190	4.006.309
L_3	2.042.122	5.169.139	2.068.905

Table 6: Confronto tra protocolli: aumento dell'interferenza, tempi espressi in nano secondi.

accedere la risorsa e l'inizio dell'esecuzione di H_2 è posticipato allungando così il tempo di completamento.

Con il taskset illustrato nella tabella 7 si intende ripercorrere il funzionamento del primo esempio forzando in questo caso il job che detiene la risorsa ad affrontare 2 migrazioni. Inoltre il fatto che vi siano 3 CPU a contendere per l'accesso della risorsa triplica il fattore per cui la sezione critica viene moltiplicata. A seconda del protocollo utilizzato aumenta l'attesa dei job accodati o il blocco subito dai job à priorità più alta.

Nella tabella 8 il comportamento è quello atteso e discusso finora, l'aspetto interessante è vedere come l'esecuzione di circa 1 ms di L_1 sia aumentato ulteriormente da 1.05 ms a 1.1 ms a causa della doppia migrazione.

Task	Partition	priority	Critical section	WCET
L_1	P_1	20	1	1
H_2	P_1	10	0	3
L_3	P_2	20	1	1
H_4	P_2	10	0	3
L_5	P_3	20	1	1

Table 7: Confronto tra protocolli: doppia migrazione.

Task	MrsP	Ceiling	Non preemption
L_1	<u>1.111.410</u>	4.312.490	<u>1.173.600</u>
H_2	3.029.214	3.131.630	4.205.578
L_3	<u>2.062.770</u>	5.301.240	<u>2.211.347</u>
H_4	3.022.036	3.099.090	5.078.436
L_5	3.030.634	6.309.370	3.184.333

Table 8: Confronto tra protocolli: doppia migrazione, tempi espressi in millisecondi.

1.2.5 Considerazioni

I risultati esaminati mettono in risalto le differenze tra i vari protocolli nella gestione della risorsa globale. Si nota come MrsP sia migliore rispetto alle alternative qui studiate: simple ceiling è inadatto a gestire la condivisione di risorse in ogni caso analizzato, mentre l'inibizione del prerilascio risulterebbe migliore in uno scenario in cui la lunghezza della sezione critica sia inferiore rispetto agli overhead dati da MrsP, in particolare il costo della migrazione.

Quest'ultimo aspetto è discusso da Burns et al. [4]: se i costi aggiuntivi dal sistema sono superiori rispetto alla lunghezza della sezione critica allora il protocollo non è più utile. Tale affermazione è confermata a seguito dell'implementazione e dei test, essi mettono in risalto come la presenza di costi aggiuntivi vada a penalizzarne le prestazioni.

Nei grafici 5, 6 e 7 sono riassunti i risultati ottenuti nell'esperimento, mettendo a confronto i tempi raccolti per ogni task nelle diverse configurazioni del sistema.

In figura 5 viene preso in considerazione L_1 , cioè il task a bassa priorità che richiede ed accede la risorsa per primo. Il grafico evidenzia come MrsP ed l'inibizione del prerilascio abbiano prestazioni molto simili, salvo il costo della migrazione nel primo caso; al contrario simple ceiling porta a risultati peggiori in ogni configurazione del sistema.

Il grafico 6 rappresenta le prestazioni raccolte di H_2 : nei sistemi gestiti tramite utilizzo di ceiling i task a priorità superiore non risentono della presenza di risorse, mentre l'inibizione del prerilascio causa notevoli ritardi nei tempi di completamento dei job in quanto subisce blocco dall'esecuzione della sezione critica.

QUESTO
ANDREBBE
STUDIATO IN
UN ESPERIMENTO
APPARTATO

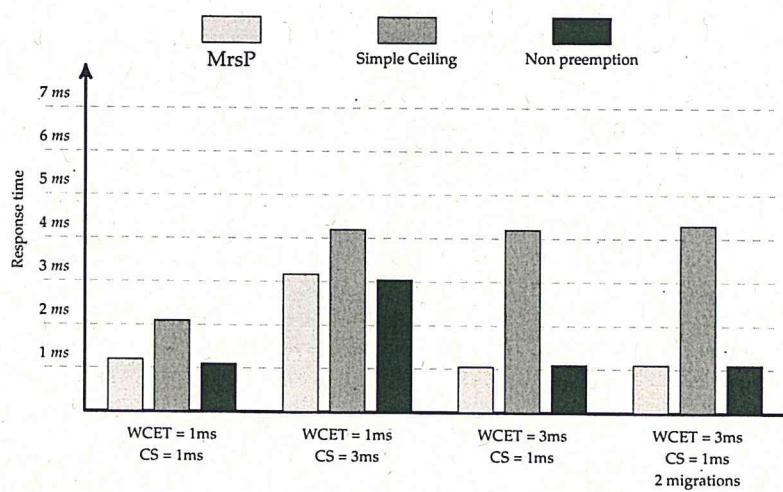


Figure 5: Response time di L_1

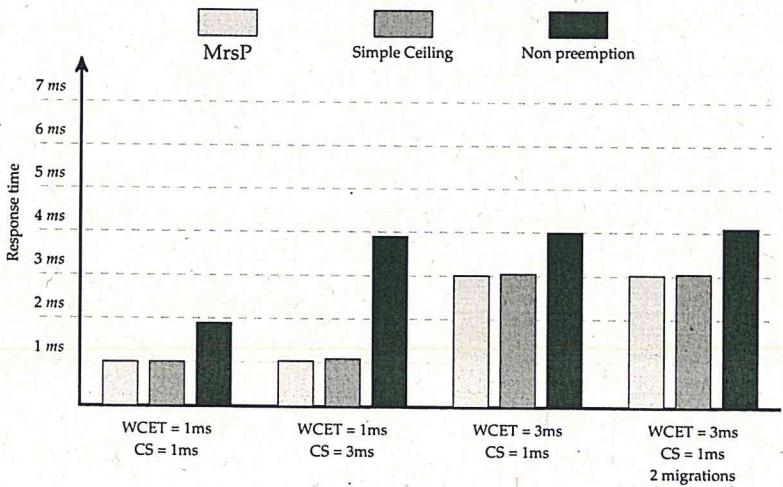


Figure 6: Response time di H_2

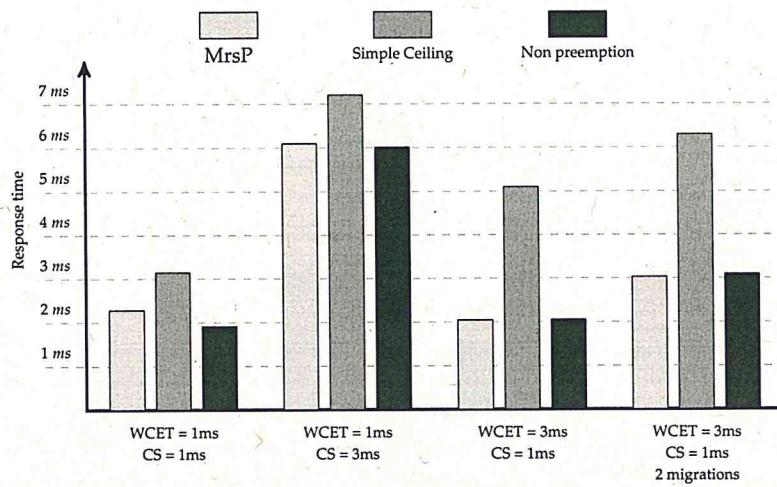


Figure 7: Response time di L_3

Infine 7 evidenzia come il job L_3 subisca l'interferenza a cui è soggetto L_1 : MrsP e non preemption tentano di minimizzare tale interferenza, di conseguenza anche L_3 giova di questi meccanismi anche se con il primo protocollo si notano i costi dovuti dalla migrazione. Nel caso di simple ceiling il job in questione incorre nell'interferenza che avviene nella prima CPU oltre che nei tempi di attesa dovuti dallo smaltimento della FIFO.

L'esperimento illustrato in questa sezione è stato tratto dai test simulati da Burns e Wellings in [4]. L'implementazione sviluppata e valutata in questo lavoro di tesi rispecchia ciò che affermano. In particolare si evidenzia come MrsP sia una buona combinazione dei vantaggi degli altri approcci esaminati in questo esperimento. Inoltre una reale implementazione al contrario della simulazione permette di ottenere un migliore riscontro dei costi aggiuntivi dati dalle primitive e, soprattutto, dalle migrazioni.

1.3 Calcolo degli overhead

MrsP combina approcci differenti in parte tratti da protocolli esistenti, i quali, nonostante alcuni aspetti positivi, hanno un funzionamento inapplicabile ad un sistema real-time oppure hanno una complessità che a livello teorico sembra ragionevole ma che in un ambiente reale non è sostenibile. Uno degli obiettivi di questo lavoro di tesi è quello di dimostrare che è possibile implementare il protocollo di Burns et al. [4] a partire da uno scheduler P-FF con un sovraccarico ragionevole del sistema.

1.3.1 Esperimento ~~NUMBER~~

Gli esperimenti esposti in questa sezione mirano a valutare l'implementazione e le scelte algoritmiche, i relativi overhead vengono studiati per capire quali task ne risentono ed in quali circostanze.

Lo sviluppo del protocollo a partire dall'implementazione fornita da LITMUS^{RT} di *partitioned fixed priority* ha reso necessario modificare alcune primitive per integrare il protocollo di accesso. Quelle prese in considerazioni sono le seguenti:

- creazione della risorsa;
- richiesta di accesso;
- rilascio della risorsa;
- chiusura della risorsa;
- operazione di schedule;
- *finish-switch*.

1.3.2 Configurazione

Il sistema non permette di avere un release time unico per tutti i job a meno di creare un taskset caratterizzato da periodi armonici, ma in questo caso risulterebbe difficile riuscire a testare alcuni meccanismi che solamente in casi particolari entrano in gioco. In un sistema privo di periodi armonici i task sono soggetti a release latency, di conseguenza non è possibile forzare determinate dinamiche con precisione. Per ottenere ~~del~~ campionamenti utili per ogni meccanismo che caratterizza MrsP è necessario creare un taskset sufficientemente grande in modo che si vengano a creare naturalmente le circostanze per poterne usufruire.

Il taskset è generato randomicamente con 25 job suddivisi su 4 CPU in modo il più possibile bilanciato. Per ogni CPU sono selezionati alcuni job ai quali

è aggiunta la richiesta alla risorsa condivisa. Ogni taskset prima di essere eseguito è analizzato con uno script creato per verificare che sia *feasible* andando ad applicare la *response time analysis* aumentata con il protocollo MrsP.

1.3.3 Obiettivo

Lo scopo di questo esperimento è valutare l'impatto del protocollo in termini di overhead; i campionamenti sono messi in relazione con i tempi di esecuzione ed i costi aggiunti dal sistema.

Non tutte le primitive sono interessanti per lo studio dei costi che il protocollo aggiunge a *runtime*: la creazione e la chiusura non sono presi in considerazione in quanto eseguite in fase di inizializzazione e finalizzazione della risorsa.

L'operazione di scheduling subisce un'unica modifica che permette di bloccare la coda dei job ready quando la richiesta in testa a tale coda ha priorità inferiore rispetto al ceiling locale. Ne consegue che il costo della primitiva risulta uguale o minore rispetto alla versione originale, di conseguenza è poco interessante per i campionamenti.

Più complesse sono le primitive inerenti a **lock** e **release** della risorsa e l'operazione di **finish-switch**; quello che ci si aspetta è che l'overhead aggiunto in caso di esecuzioni nella norma non sia elevato e che vada ad aumentare a seconda dei meccanismi attivati. Per esecuzione "normale" si intende quei casi in cui non vi sono interferenze al lock holder da parte di job a priorità superiore al ceiling locale, pertanto i job che contendono per il possesso della risorsa effettuano la richiesta, se occupata attendono fino ad ottenerla, eseguono la sezione critica e la rilasciano.

I costi rilevati durante i campionamenti possono influenzare il sistema a tre differenti livelli (figura 8):

- (i) il solo job che esegue la primitiva; questo accade se non si va a modificare il ceiling locale, di conseguenza non si causa blocco ai job a priorità inferiore a tale valore;
- (ii) la sola CPU in cui è allocato il job corrente; questo succede nei casi in cui i costi vengano generati in circostanze di ceiling innalzato ma non si è in possesso della risorsa;

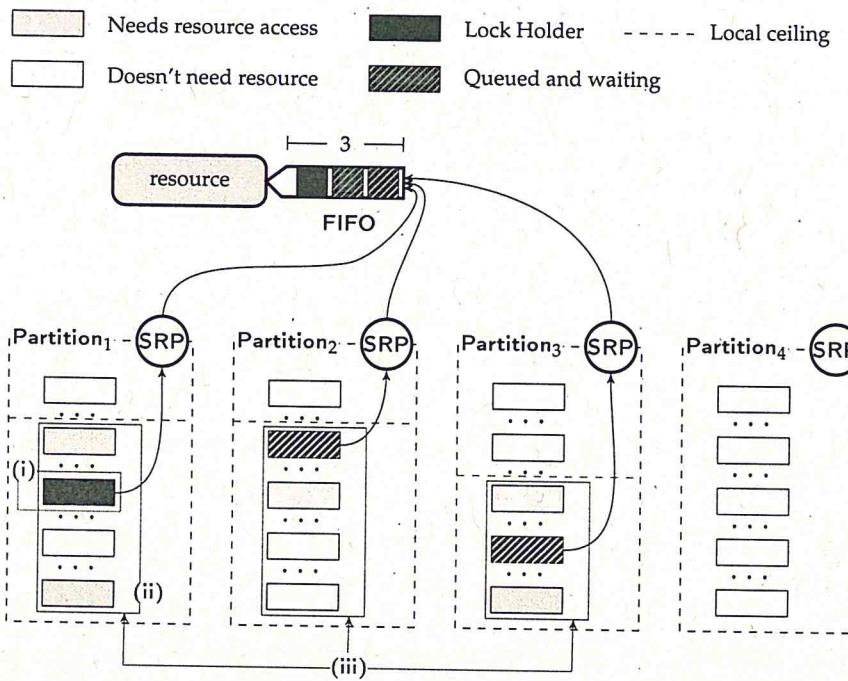


Figure 8: Overhead e relativa influenza nel sistema.

- (iii) l'intero sistema, inteso come le CPU in cui stanno eseguendo job che attendono di accedere la risorsa.

1.3.4 Risultati

I dati del sistema sono organizzati secondo la figura 9: per ogni CPU è presente una struttura dati che tiene traccia del task attualmente in esecuzione, il numero della CPU, la coda dei task ready in attesa di eseguire ed il ceiling locale; la risorsa invece è composta dal task che la detiene, la coda di task in attesa di ottenerla e la lista dei ceiling calcolati in fase di inizializzazione. L'accesso alle strutture dati da parte dei processi viene reso sequenziale ed in mutua esclusione tramite l'uso dei **spin-lock**. Quest'ultimo consiste in un particolare tipo di lock del kernel **LINUX**, lo si acquisisce tramite l'operazione `spin_lock(lock)` e lo si rilascia con `spin_unlock(lock)`. Di conseguenza le primitive che prevedono l'utilizzo delle strutture dati prima di eseguire la sezione critica devono ottenere lo spin-lock, tale operazione aggiunge ulteriori overhead oltre a quelli dettati dal protocollo in sé.

Di seguito vengono prese in considerazione le primitive, per ognuna si

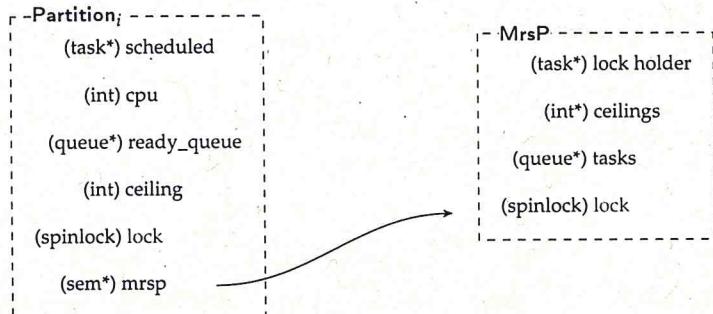


Figure 9: Strutture dati.

indicando gli overhead campionati; i valori rilevati sono espressi in nanosecondi.

Lock L'operazione di richiesta della risorsa consiste nei tre passaggi elencati di seguito e raffigurati in figura 11:

1. innalzamento della priorità del job e del ceiling locale al valore calcolato in fase di inizializzazione, cioè la priorità più alta tra tutti i job allocati in quella CPU che accedono la risorsa, accodamento della richiesta nella FIFO ed infine determina la stato corrente della risorsa ed eventuale proprietario;
2. in base alle informazioni ottenute al passo precedente, se il proprietario non è in esecuzione, cioè è accodato nella coda dei job ready di un altro processore, gli viene concesso di eseguire nella CPU corrente, viene perciò effettuata una migrazione;
3. se il job che ha inoltrato la richiesta non ha ottenuto la risorsa effettua attesa attiva fino ad arrivare in testa alla coda FIFO.

La prima fase causa un overhead pari a circa 800 ns ~~e~~ il costo principale riguarda le operazioni sulla coda di richieste. Questo costo è intrinseco alla risorsa e viene pagato dal job stesso e da quelli della stessa CPU. Tale influenza è motivata dal fatto che la prima operazione che viene effettuata è l'innalzamento di priorità e del ceiling, di conseguenza il tempo di blocco oltre alla sezione critica comprende anche questo overhead.

I dati utili a questa operazione sono condivisi con altri processi, di conseguenza l'accesso deve essere gestito tramite il sistema di spin-lock in

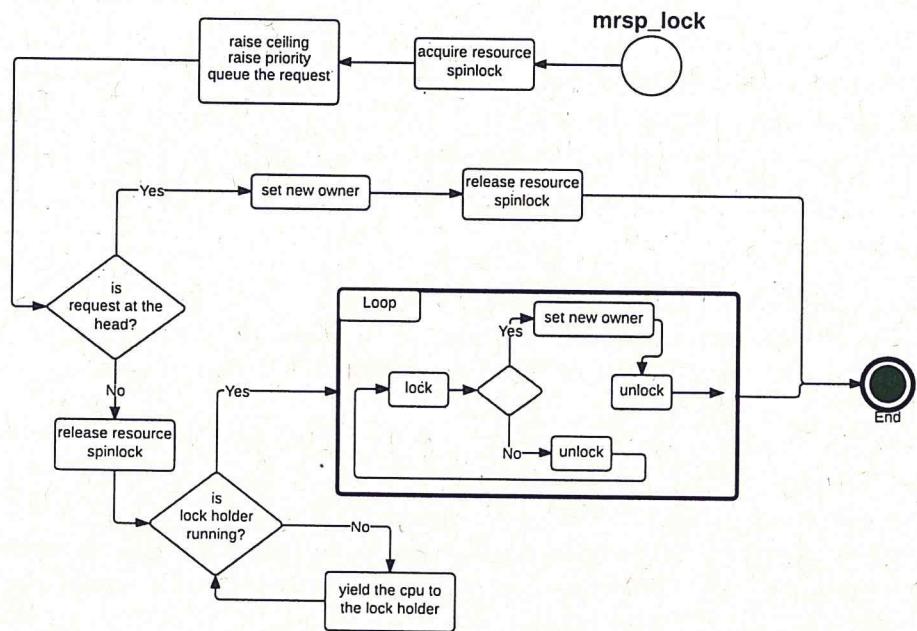


Figure 10: Lock: diagramma di flusso.

modo da garantire mutua esclusione ed evitare stati inconsistenti causati da accessi in parallelo da distinte CPU. Quindi per eseguire i passaggi indicati in questa fase è necessario prima di tutto aver acquisito lo spin-lock della risorsa aumentando il tempo necessario per effettuare la richiesta di accesso. Il suo costo va ad influire solamente sul job in questione ed il suo ammontare dipende dal numero di processori che contendono: un maggior numero di richieste in parallelo significa maggior interferenza, quindi un incremento dell'attesa.

La seconda fase consiste nel togliere il job dalla coda ready della CPU in cui si trova per poi modificarne la priorità ed accodarlo in quella corrente. In questo modo è possibile selezionarlo per l'esecuzione alla successiva operazione di scheduling. Le due singole operazioni sulle code richiedono in media un tempo pari a 500 ns. Anche in questo caso il sistema è dotato di un sistema di lock che mirano a serializzare le operazioni, quindi è necessario ottenere lo spin-lock prima di una CPU e poi, dopo aver rilasciato il precedente, di quella corrente. Le misurazioni del tempo complessivo della seconda fase indicano che le tempistiche medie sono vicine ai 2k ns, e rispecchiano la contesa per ottenere il lock sulle CPU.

Il costo della migrazione è in media pari a **6k ns**, esso consiste nel tempo che impiega la CPU corrente ad effettuare il context switch tra il job che richiede la risorsa e quello che la detiene. Tale cambio di job in esecuzione viene effettuato tramite un'operazione di scheduling; i strumenti di misurazione delle primitive di sistema confermano che il tempo campionato influenza il costo di tale operazione. COSA VUO DIRE?

Gli overhead identificati influiscono sull'intero sistema in quanto l'esecuzione della sezione critica da parte del job che detiene la risorsa non riprende immediatamente non appena vi è una nuova CPU disponibile, bensì dopo una quantità di tempo dettata principalmente dalla migrazione.

determinata

La terza ed ultima fase della primitiva di richiesta della risorsa consiste nell'eseguire attesa attiva. Essa consiste nell'eseguire ciclicamente un controllo alla testa della coda FIFO, e nel caso sia il turno del job corrente acquisisce la risorsa. Ogni ciclo ha un costo di **500 ns**, anche in questo caso dettato dalle operazioni sulla coda; questo dato non tiene conto dell'overhead per ottenere lo spin-lock della risorsa.

Quest'ultimo caso è un costo intrinseco del protocollo e dell'approccio spin-based, di conseguenza non aggiunge alcun costo, salvo un eventuale offset dato dal tempo che il job impiega ad accorgersi che è il suo turno. In tale circostanza si va ad allungare i tempi di attesa degli altri job contendenti, quindi anche il tempo di blocco subito nelle CPU.

Release Al momento del rilascio della risorsa vengono effettuate le seguenti operazioni:

1. il job rilascia la risorsa, ripristina la propria priorità ed il ceiling della CPU in cui è allocato e toglie la richiesta dalla testa della coda;
2. se la FIFO non è vuota, viene controllato lo stato del job in testa, se non sta eseguendo si cerca una CPU in cui farlo eseguire;
3. il job rilascia la CPU in cui sta eseguendo se non si trova nella propria di origine.

Le operazioni della prima fase causano un overhead di **500 ns**; come nelle circostanze precedenti ad influire sono le manipolazioni della FIFO. In questo modo si posticipa l'esecuzione del prossimo job in testa alla coda, di conseguenza influenza ogni CPU in attesa di accesso.

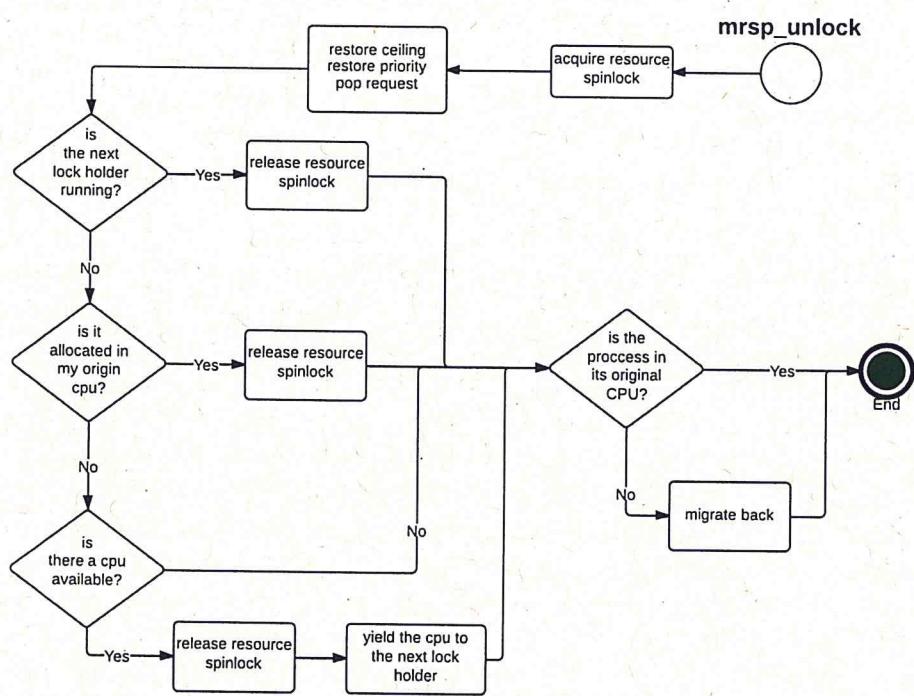


Figure 11: Lock: diagramma di flusso.

Come nel caso dell'acquisizione della risorsa, in alcune circostanze è necessario forzare il prossimo lock holder a migrare in un'altra CPU per proseguire nell'esecuzione e limitare i tempi di attesa della coda FIFO. I campionamenti hanno evidenziato le medesime tempistiche: **2k ns** per le operazioni eseguite tenendo in considerazione i tempi causati dai spin-lock sulle CPU e **6k ns** per effettuare la migrazione vera e propria. L'overhead viene pagato dall'intero sistema.

Nell'ultima fase il job che ha rilasciato la risorsa, se necessario, ritorna alla propria CPU di origine. Questo viene effettuato andando ad abbassare la priorità del job al di sotto del ceiling e forzando lo scheduler a selezionare la testa della coda ready.

Tale operazione raccoglie gli overhead delle seguenti operazioni:

1. parte della chiamata di sistema per rilasciare la risorsa;
2. schedule nella CPU attuale;
3. operazione di post-schedule, la quale riaccoda il task nella propria CPU;
4. schedule nella CPU di origine.

Questa operazione di conseguenza risulta molto costosa, circa **65k ns**, e viene pagata solamente dal job che ha effettuato il rilascio. Tale attribuzione consegue dal fatto che la risorsa già nella prima fase è stata rilasciata, quindi potenzialmente il prossimo job può eseguire la propria sezione critica, ed il valore di ceiling della CPU di origine è stato ripristinato, permettendo ai job a priorità inferiore di eseguire nuovamente.

Finish-switch Al contrario delle altre primitive, non vi è una sequenza di fasi, bensì, se necessario, si aziona solamente un meccanismo tra quelli implementati nella primitiva:

- meccanismo base di migrazione di MrsP in caso di prerilascio;
- meccanismo di notifica di CPU di nuovo disponibile
- meccanismo di migrazione di LITMUS^{RT}

Il primo prevede di cercare una CPU disponibile per la migrazione, il suo costo dipendente principalmente dalla lunghezza della coda. In caso di successo, il job modifica la propria CPU di riferimento e la priorità in base al nuovo ceiling, infine viene accodato nella CPU disponibile, forzando su

di essa un'operazione di scheduling. Questo insieme di operazioni risulta onerosa in quanto opera sulla coda e necessita dello spin-lock della risorsa e della coda ready. Il campionamento ha evidenziato un caso medio di circa 24k ns.

Inoltre tali operazioni generano una migrazione: al contrario dei casi precedenti il job non è accodato, bensì è in uno stato di prerilascio. Il costo campionario è pari a 37k ns.

Nel secondo caso viene inizialmente elaborato lo stato della CPU e del lock holder, se quest'ultimo non è in esecuzione e la CPU corrente è tra quelle accodate nella FIFO della risorsa si ricorre alla migrazione per cedere l'esecuzione. I campionamenti indicano che la prima parte ha un costo di 3k ns, questo in quanto si necessita di mutua esclusione durante la manipolazione della coda, quindi di acquisire lo spin-lock. Il costo della migrazione è in linea con gli altri casi in cui il job preso in considerazione è accodato: circa 6k ns.

Gli overhead campionati che caratterizzano la primitiva di finish-switch affliggono il job mentre detiene la risorsa, quindi i ritardi provocati penalizzano l'intero sistema.

1.3.5 Considerazioni

Questo esperimento e gli overhead riportati evidenziano come MrsP abbia un costo relativamente basso nella maggior parte dei casi, cioè quando non vi sono circostanze in cui i vari meccanismi di migrazione entrano in gioco.

I strumenti di misurazione permettono di avere un campionamento delle operazioni nella loro interezza, cioè da quando viene effettuata la *system call* a quando si è conclusa. Tali misurazioni hanno evidenziato come le primitive di LOCK e UNLOCK della risorsa comportino un overhead di circa 2k ns, di cui solamente 500 / 800 ns sono riportabili all'implementazione di MrsP. Questo permette di affermare nuovamente che se non vi sono prerilasci il protocollo ha un impatto relativamente basso sul sistema.

Considerazioni differenti vanno fatte in caso di prerilascio: l'operazione più onerosa è data dalla migrazione al momento del rilascio della risorsa, essa accade solamente una volta ed il suo overhead rispecchia come si vadano ad accumulare una serie di costi di sistema e relative primitive prima che il job riprenda ad eseguire nella propria CPU.

Gli altri meccanismi di migrazione al contrario sono meno onerosi, hanno però lo svantaggio che non si può stimare quante volte verranno innescate nell'arco di una sezione critica dato che dipendono dalle dinamiche del sistema durante l'esecuzione della sezione critica.

Le tempistiche campionate sono rappresentate graficamente nel grafico 12, in esso viene tralasciata la migrazione al momento del rilascio in quanto affligge solamente il job che la deve effettuare senza influenzare altri task.

Alla luce di queste considerazioni, come accennato in precedenza, MrsP risulta utile in quelle situazioni in cui la somma degli overhead hanno un valore inferiore rispetto all'interferenza che viene causata dai job a priorità superiore, di conseguenza dipende dalla lunghezza della sezione critica. Se tali costi fossero superiori è conveniente un approccio basato sull'inibizione del prerilascio in quanto gli overhead pagati dal sistema sarebbero superiori ai benefici tratti dai job a priorità più alta del ceiling.

→ INSERIRE TABEUA RIASSUNTIVA VANTAGGI E SVANTAGGI

1.4 Confronto in assenza di risorsa

In un sistema real-time la condivisione di risorse globali tra task allocati su differenti CPU è un caso particolare, di conseguenza risulta interessante un confronto tra l'implementazione del protocollo con uno scheduler che non le prevede nell'esecuzione di un taskset in cui i task sono indipendenti.

1.4.1 Esperimento

Gli esperimenti discussi in questa sezione pongono a confronto l'implementazione di partenza di *partitioned fixed priority* fornita da LITMUS^{RT} con la versione modificata per integrare MrsP.

1.4.2 Configurazione

I taskset sono generati casualmente con alcuni accorgimenti: i periodi sono armonici e variano tra i 25ms ed i 200ms, i tempi di esecuzione sono calcolati con distribuzione di probabilità uniforme tra 0.1 e 0.4 in relazione al periodo, l'esecuzione dell'ultimo task creato viene ridimensionata in base alle esigenze per ottenere un'utilizzazione dell'intero taskset pari al valore richiesto. Il problema del *bin-packing* per allocare i task nei processori segue l'euristica *worst-fit*. Infine ogni esecuzione ha una durata di 15 secondi.
osserva

Il valore di utilizzazione del taskset è pari alla somma del rapporto tra WCET e periodo di ogni task ed il carico di lavoro di sistema. L'esperimento

QUESTO VALORE
UNITE VA INDICARE
ESPLICATAMENTE

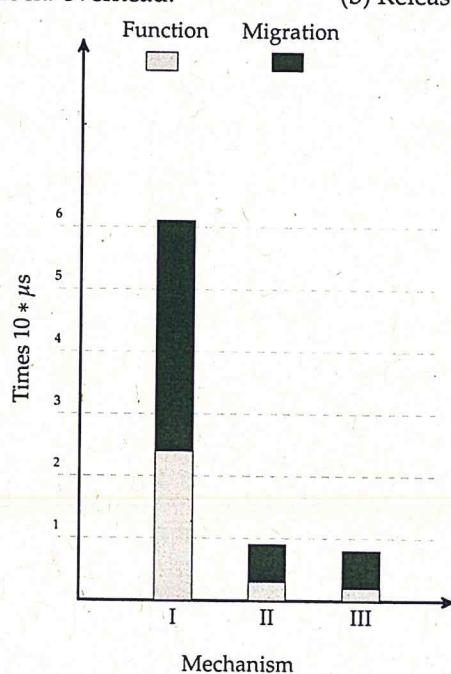
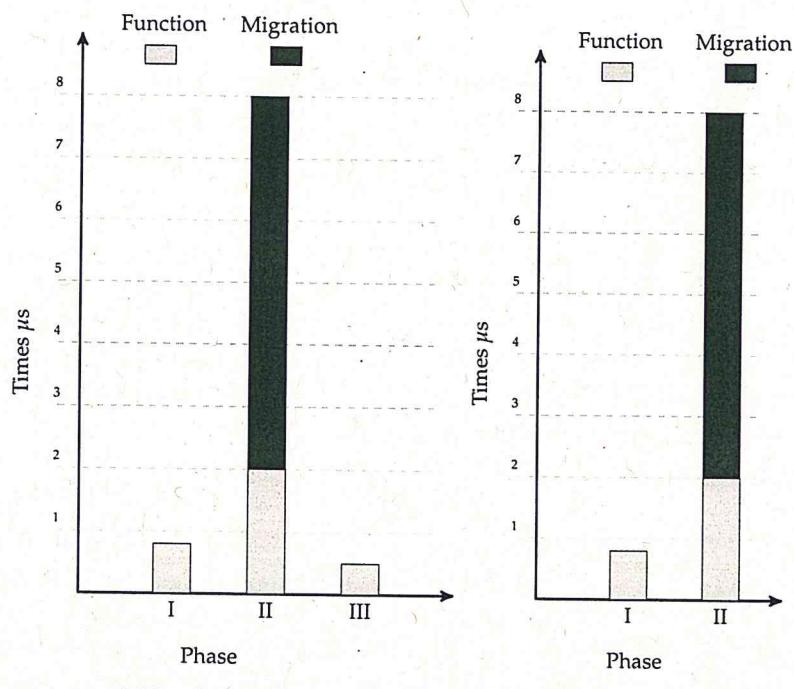


Figure 12: Overhead di sistema delle primitive.

prevede che i taskset siano creati in modo tale da rispettare una soglia di utilizzazione che varia dal 50% del carico totale fino al 100%, nel caso di un sistema composto da 4 processori le soglie saranno 2.0, 2.4, 2.8, 3.0, 3.2, 3.4, 3.6, 4.0.

1.4.3 Obiettivo

L'obiettivo è quello di studiare le esecuzioni dei taskset ed il comportamento degli scheduler all'aumentare del fattore di utilizzazione. Quello che ci si aspetta è che l'overhead aggiunto dall'integrazione di MrsP non vada ad intaccare il normale svolgimento di *partitioned fixed priority*, ma ad alti livelli di sovraccarico del sistema anche i minimi costi possono causare deadline miss che altrimenti non avverrebbero, questo considerando che già il sistema di per sé aggiunge dei ritardi dati dalle singole primitive.

La scelta di incrementare il livello campionamento per carichi vicini all'80% è data dal comportamento atteso: l'assegnamento della priorità di ogni singolo task è stato effettuato utilizzando la tecnica di *rate monotonic assignment*, di conseguenza possiamo far uso della *utilization-based analysis*, secondo cui un valore di utilizzazione inferiore al 0.69 è condizione sufficiente ma non necessaria per ottenere un taskset schedulabile (Liu et al. [5]). Di conseguenza se il protocollo aggiunge elevati costi rispetto ad un'implementazione che non lo supporta ci si aspetta che sia nell'intorno di questa soglia che i due scheduler abbiano comportamenti diversi per quanto riguarda il numero di *deadline miss*.

SOL SE I
TASK SONO
INDEPENDENTI !!

1.4.4 Risultati

Dalle esecuzioni sono stati estrapolati diverse informazioni: il numero di deadline miss rappresentato nel grafico 13 indica come il comportamento delle due implementazioni sia il medesimo. Nel grafico non sono stati riportati i dati relativi a workload pari al 90% ed al 100% in quanto già al 90% sei taskset su dieci producevano deadline miss su entrambi; essendo i periodi armonici, ad ogni "iper-periodo" le stesse "dinamiche" sono ripetute fino alla fine dell'esecuzione, pertanto ci si aspetta un numero molto alto. Di conseguenza gli overhead degli scheduler non influiscono sui dati rilevati.

Le primitive principali utilizzate dal protocollo di accesso sono state discusse in precedenza; l'implementazione di *fixed priority scheduler* in assenza di risorsa condivisa non richiede le operazioni di accesso e rilascio, ergo sono state prese in considerazione l'evento di dispatching, di *context switch*

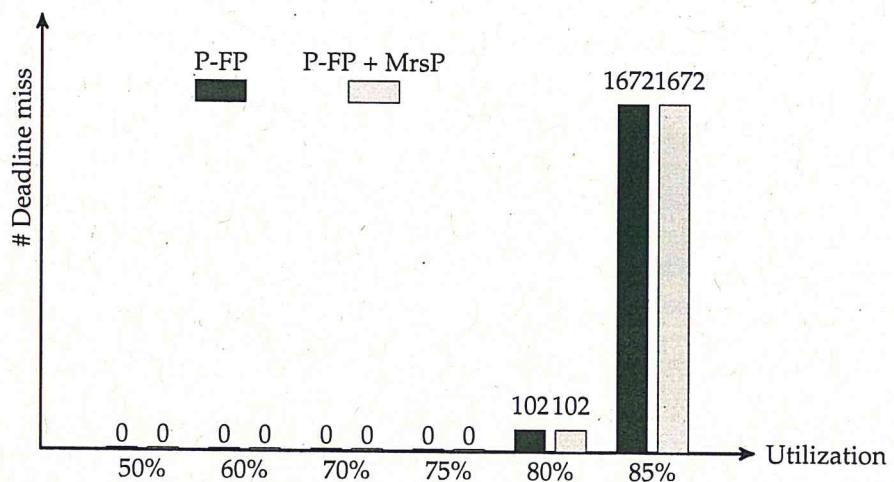


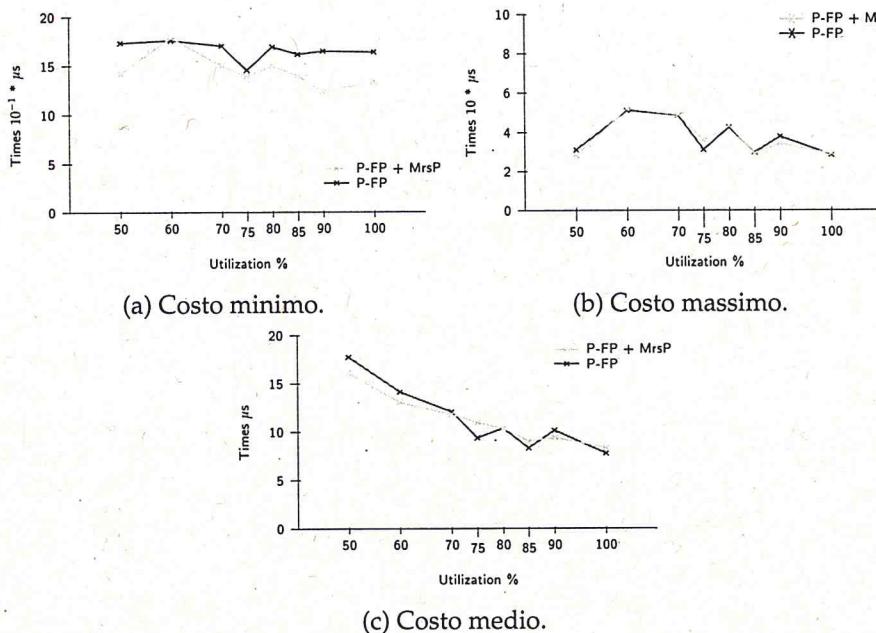
Figure 13: Numero di *deadline miss*.

e di *job release*.

L'operazione di dispatching ha il compito di decidere a quale job assegnare l'esecuzione nel processore, consiste quindi nel determinare lo stato del job corrente e se necessario prerilasciarlo a favore di quello in testa alla coda dei job pronti all'esecuzione. Tale primitiva viene richiamata in molteplici circostanze ed il job in coda acquisisce la CPU quando è inutilizzata o ha priorità superiore rispetto a quello in esecuzione. Nelle figure 14a, 14b e 14c sono rappresentati e confrontati rispettivamente costo minimo, massimo e medio della primitiva di dispatching. In tutti e tre i grafici si nota come le due implementazioni abbiano lo stesso comportamento e la differenza tra i valori ottenuti è bassa, soprattutto per quanto riguarda il costo massimo.

Nei grafici 15a, 15b e 15c i rilevamenti dell'operazione di release mostrano nuovamente un andamento molto simile tra i due scheduler anche se con valori leggermente differenti. Tali divergenze non sono comunque riportabili all'implementazione in quanto tale primitiva non viene modificata, inoltre le normali dinamiche di *partitioned fixed priority* non sono alterate. All'evento di release lo scheduler ha il compito di inserire il job nella coda ready, in questo caso mantenendo l'ordinamento in base alla priorità.

L'operazione di post-schedule è fondamentale nell'integrazione di MrsP, in essa vengono attuati meccanismi che permettono al job che detiene la risorsa di migrare quando necessario, al contrario in assenza di risorsa essa non ha alcun particolare utilizzo. Il costo minimo, massimo e medio risulta in linea



PROVA
SPIEGLARE
I DENTI

Figure 14: Confronto tra implementazioni: *schedule*.

con l'implementazione che non prevede condivisione di risorse: figure 16a, 16b e 16c.

1.4.5 Considerazioni

L'esperimento ha lo scopo di mettere a confronto il medesimo *scheduler* con e senza l'integrazione con *MrsP*.

I dati raccolti mostrano come nel caso di assenza di risorse globali non vi sia differenza tra le due, questo è un risultato importante in quanto la condivisione di risorse tra job allocati in diversi processori è un caso particolare e non quello normale. Di conseguenza possiamo affermare che gli overhead del protocollo di accesso non hanno un impatto sul sistema quando non viene utilizzato.

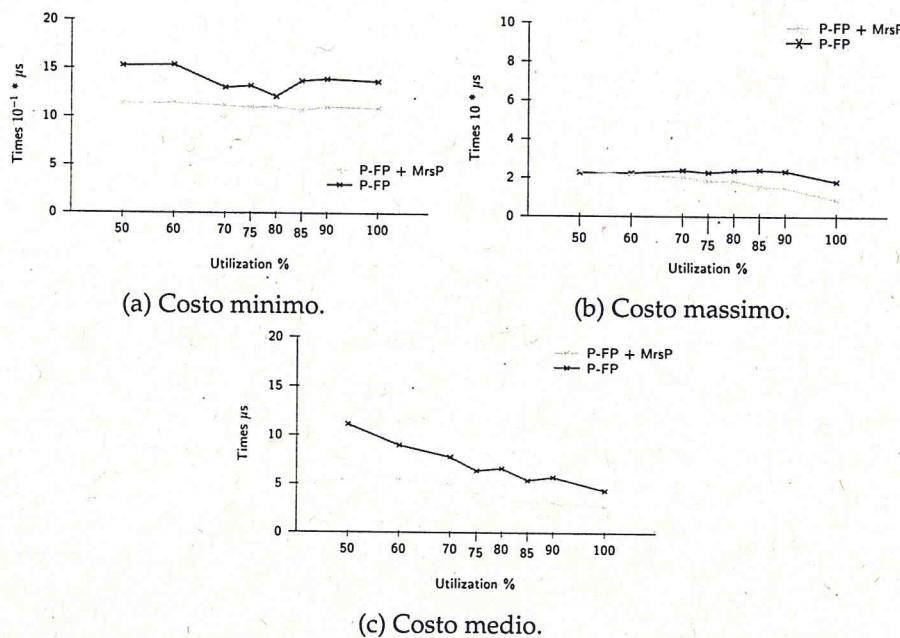


Figure 15: Confronto tra implementazioni: *job release*.

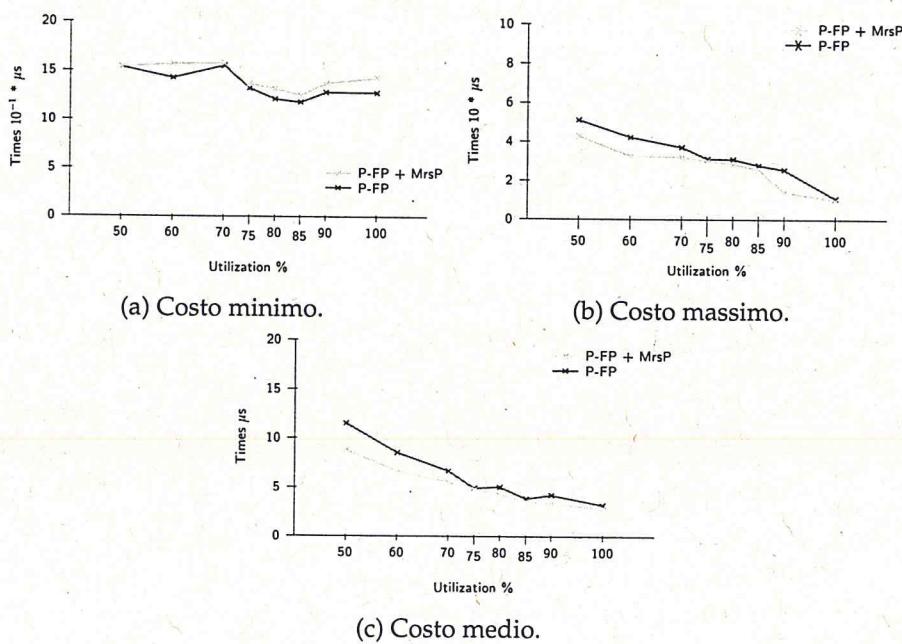


Figure 16: Confronto tra implementazioni: *context switch*.

Appendices

A LITMUS^{RT}

Di seguito viene fornita una panoramica di approfondimento del plug-in LITMUS^{RT} e del suo funzionamento.

LITMUS^{RT} è una patch per aggiungere ad un kernel Linux un'estensione real-time, in particolare per creare uno scheduler multiprocessore real-time con supporto per la sincronizzazione. Il kernel viene modificato per supportare un modello di task periodici e plug-in modulari per creare scheduler. Esso fornisce supporto, nonché diverse implementazioni, per scheduler globali, partizionati e cluster.

Il suo obiettivo principale è quello di fornire una piattaforma di sviluppo e sperimentazione nell'ambito dei sistemi real-time. Esso mette a disposizione una serie di funzioni che rispondono a determinati eventi per creare un prototipo di scheduler. Tali funzioni sono elencate e descritte nella tabella 9. Per coerenza con la piattaforma in questione in alcuni casi al posto del termine job o processo viene utilizzato task.

Tutte le funzioni sono opzionali, salvo `schedule()` e `complete_job()` per le quali è obbligatorio fornire un'implementazione, negli altri casi viene fornita una versione di default in caso di mancato utilizzo.

Le funzioni principali sono `schedule()`, `tick()` e `finish_switch()`, esse rispondono ad eventi di scheduling:

- `schedule()` viene richiamata quando il sistema necessita di selezionare il task da eseguire, a livello logico consiste in un *context switch* e viene eseguita nel processore in cui è stata richiesta;
- `tick()` viene richiamata ogni quanto di tempo e se necessario richiama la funzione di `schedule()`, è utile per quei scheduler basati su quanti di tempo, per esempio PFair;
- `finish_switch()` viene richiamata dopo ogni *context switch*, utile se lo scheduler prevede un particolare comportamento da parte del task che interrompe la propria esecuzione.

Un secondo insieme di funzioni mira a gestire gli eventi che caratterizzano