Corso di laurea in Ingegneria Informatica

Indirizzo Sistemi e Applicazioni Informatici

Studio e implementazione di un algoritmo di schedulazione real-time per il risparmio energetico sul sistema Linux

Claudio Scordino

Anno accademico 2002/2003

Relatori:

Prof. Paolo Ancilotti

Prof. Giuseppe Lipari

Prof. Marco Di Natale

Università degli Studi di Pisa

Indice

Alla mia famiglia

Introduzione

L'importanza di un sistema real-time

L'evoluzione della tecnologia apre continuamente nuovi orizzonti nel panorama scientifico: oltre ad una informatizzazione di settori che non lo erano, l'Information Technology permette oggi di vedere frontiere considerate irraggiungibili, o addirittura impensabili, fino a pochi anni fa.

All'interno di numerosi sistemi elettronici (spaziando da un semplice telefono cellulare ad un complesso sistema di controllo di una centrale nucleare) è entrato con prepotenza il concetto di tempo reale (o *real-time*). Un sistema real-time è un qualsiasi sistema che deve reagire ad eventi (interni o esterni) entro un tempo prestabilito. Il sistema è in grado di offrire garanzie precise sui tempi di risposta, e perciò la sua evoluzione temporale è prevedibile a priori. Questo concetto risulta fondamentale in tutti quei sistemi in cui è necessario avere la risposta del sistema entro un determinato periodo di tempo.

Purtroppo, nonostante negli ultimi decenni sia stata sviluppata una teoria matematica che permette di formalizzare in modo preciso il comportamento di un sistema real-time, spesso ancora oggi la progettazione di questi sistemi viene fatta in modo empirico, affidandosi a semplici calcoli euristici.
Altre volte, il sistema viene dimensionato nell'ipotesi del caso peggiore (*worst case*), risultando così in uno sfruttamento solo parziale delle risorse da esso offerte.
La progettazione viene impostata considerando che un sistema sufficientemente veloce è in grado di rispondere in modo soddisfacente in qualunque

condizione. Tuttavia, bisogna osservare che maggiore è la velocità del sistema (ovvero la sua potenza computazionale), maggiore è anche il suo consumo di risorse. Questo può rappresentare un problema nel campo dei *sistemi Embedded*, nel quale è fondamentale ridurre il più possibile le risorse necessarie al sistema.

La creazione di sistemi operativi per sistemi embedded a partire da quelli esistenti per macchine di più alto livello, si è spesso scontrata con l'impossibilità di avere a disposizione un numero elevato di risorse.
In un sistema embedded è molto importante il trade-off tra la potenza computazionale offerta e le risorse consumate per ottenerla. In particolare, risultano determinanti risorse quali le *dimensioni*, *il consumo energetico* ed il *costo*.

Il problema del consumo

Un altro problema che con il trascorrere degli anni sta diventando sempre più rilevante è quello del *consumo*. Il concetto di consumo è molto generico, e può riguardare l'esaurimento delle batterie (come ad esempio in un computer palmare) oppure essere legato ad un problema di surriscaldamento (come ad esempio nella CPU di un Personal Computer).
All'interno dell'insieme dei sistemi embedded, il problema del consumo energetico è particolarmente importante per i sistemi *autonomi*, ossia sistemi che hanno 'a bordo' una batteria che deve durare più a lungo possibile.

L'aumento della potenza di calcolo dei processori odierni è generalmente dovuto all'innalzamento della loro frequenza di clock, che porta ad un maggiore consumo di energia, e ad una maggiore dissipazione di calore.
La soluzione che generalmente viene adottata è quella di offrire la possibilità di cambiare dinamicamente la frequenza di clock del sistema, in accordo al lavoro che esso deve svolgere. Ciò permette di ridurre il consumo della CPU in tutti quei momenti in cui il carico che essa deve gestire è sufficientemente basso.

Obiettivo

Questa tesi si pone l'obiettivo di realizzare un sistema operativo soft real-time con caratteristiche di risparmio energetico, per sistemi general purpose. In altre parole, vogliamo realizzare un sistema operativo che dia delle garanzie sull'esecuzione dei singoli processi, e che permetta di ridurre il consumo al minimo livello necessario per soddisfare queste garanzie.

La tesi è stata svolta nell'ambito del progetto OCERA (*Open Components for Embedded Real-time Applications*), il cui obiettivo principale è la progettazione e l'implementazione di una libreria di software open source per la realizzazione di sistemi embedded real-time. Questi componenti verranno usati per creare un sistema flessibile (le nuove politiche di scheduling supporteranno un'ampia varietà di applicazioni), configurabile (scalabile da un sistema piccolo ad uno pieno di caratteristiche) e portabile (adattabile a diverse configurazioni hardware e software). Lo scopo del progetto è quello di portare all'interno del mondo industriale una tecnologia real-time innovativa, basata su risultati scientifici riconosciuti e dimostrabili formalmente.

Partendo dall'implementazione quasi definitiva dell'algoritmo *Constant Bandwidth Server* ([Abe98]), è stato implementato un algoritmo migliore, *Greedy Reclamation of Unused Bandwidth*([Lip00]), ed esteso in modo da supportare il risparmio energetico.

Organizzazione del documento

Il presente documento è organizzato nel modo seguente.

Il capitolo 1 richiama le proprietà e le caratteristiche di un sistema real-time, ed introduce il modello di un task real-time. Viene affrontato il problema della schedulazione (o *scheduling*), e vengono descritti i principali algoritmi che permettono di ottenere una schedulazione fattibile. Viene inoltre motivata la scelta di un algoritmo a priorità dinamiche (*Earliest Deadline First*) rispetto ad un algoritmo a priorità statiche (*Rate Monotonic*).

Il capitolo 2 illustra l'algoritmo CBS (*Constant Bandwidth Server*), al quale si rifanno gli algoritmi proposti nei successivi capitoli (e che ne rap-

presentano una estensione). Il punto di partenza di questa tesi è, infatti, l'implementazione dell'algoritmo nell'ambito del progetto OCERA([Abe02]).

I capitoli 3 e 4 sono dedicati ad alcune estensioni dell'algoritmo CBS, che permettono di superare i principali limiti intrinseci dell'algoritmo stesso. Il capitolo 3 descrive l'algoritmo GRUB (*Greedy Reclamation of Unused Bandwidth*).
Il capitolo 4 illustra una modifica all'algoritmo GRUB che permette di ridurre il tempo di risposta del sistema nei confronti di un task; viene inoltre dimostrata la correttezza di questa nuova soluzione.

Il capitolo 5 affronta il problema di come realizzare un sistema operativo real-time. Viene motivata la scelta di *Linux* come sistema operativo di partenza, e vengono descritte dettagliatamente tutte le strutture dati usate per implementare il nostro sistema real-time.

Il capitolo 6 mostra l'implementazione dell'algoritmo di scheduling CBS (descritto nel capitolo 2) e descrive tutte le modifiche necessarie per trasformare questa implementazione nell'algoritmo GRUB. Viene inoltre mostrata una possibile implementazione della regola di *Hard Reservation* [Raj98] (descritta nel capitolo 4).

Il capitolo 7 affronta il problema del risparmio energetico, descrivendo una implementazione per ridurre il consumo di energia su un processore Intel PXA250.

Il capitolo 8, infine, mostra come compilare ed installare lo scheduler, e descrive la serie di test che sono stati svolti per verificarne il corretto funzionamento.

Il codice completo dello scheduler real-time è reperibile al seguente URL:

http://www.ocera.org/download/components/

La tesi si riferisce alla versione 1.0 del progetto OCERA.

Capitolo 1

I sistemi real-time

1.1 Cosa è un sistema real-time

Consideriamo il sistema di controllo di un certo ambiente. L'ambiente crea da sè alcuni eventi, ed il sistema di controllo ha il compito di rispondere a questi eventi modificando l'ambiente entro un determinato intervallo di tempo. Non è sufficiente che il sistema di controllo sia corretto: deve anche essere in grado di rispondere in tempo.

Un *sistema real-time* può essere definito come un *sistema capace di garantire i vincoli temporali dei processi sotto il suo controllo a partire da alcune ipotesi sul loro comportamento e dal modello dell'ambiente esterno.* Tipicamente, quindi, un sistema real-time è un sistema di controllo, che gestisce e coordina le attività di un sistema controllato (che può essere visto come l'ambiente con il quale il sistema real-time interagisce). L'interazione è bidirezionale (attraverso sensori e attuatori), e deve rispettare alcuni vincoli temporali.

Il concetto di 'tempo reale' non è un sinonimo di velocità: nella computazione real-time è sufficiente che il sistema risponda *entro i vincoli temporali.* Il termine 'reale' implica che il clock dell'ambiente e quello del sistema di controllo devono essere sincronizzati. Dunque, la proprietà più importante che un sistema real-time deve possedere non è la velocità, ma è la *prevedibilità.* Questo *determinismo* permette di dare delle *garanzie* sull'evoluzione del

sistema (in particolare, un sistema operativo real-time deve essere in grado di determinare l'istante di completamento dei processi in modo abbastanza preciso). La *rapidità* del sistema rimane comunque una caratteristica desiderabile, perché permette al sistema di rispondere con bassa latenza agli eventi che necessitano di una attenzione immediata.

Un'altra caratteristica desiderabile in un sistema real-time è la possibilità di far coesistere processi ordinari e processi real-time. A differenza dei processi ordinari, ogni processo real-time ha bisogno di rispettare delle scadenze temporali (*deadline*) entro le quali deve poter eseguire per una quantità sufficiente di tempo. Possiamo suddividere i sistemi real-time in due grandi classi di appartenenza: *Hard* real-time e *Soft* real-time.

I sistemi hard real-time rappresentano tutti i sistemi in cui non è possibile non rispettare queste scadenze temporali (evento chiamato *deadline miss*); per questo motivo, i sistemi hard real-time non possono sfruttare la prestazione del caso medio per compensare quella del caso peggiore. Generalmente questi sistemi hanno il compito di controllare o monitorare qualche dispositivo fisico.

Nei sistemi soft real-time, invece, il processo non è caratterizzato da una scadenza 'rigida' e può essere completato anche oltre il limite di tempo specificato (infatti si ha soltanto un degrado delle prestazioni). Ad esempio, è questo il caso in cui più utenti condividono la stessa risorsa, ed il sistema real-time deve garantire una determinata qualità del servizio ad alcuni utenti.

I settori applicativi in cui risulta rilevante la presenza di un sistema real-time sono numerosi. Tra essi possiamo citare:

Hard:

- militari (es. sistema missilistico)
- controlli industriali (robotica)
- centrali chimiche e nucleari

Soft:

- telecomunicazioni (es. centrali a switch)

- sistemi multimediali
- simulatori di volo

1.2 Scheduling

Il compito di un sistema real-time è quello di gestire e controllare l'assegnamento delle risorse (*scheduling*) alle varie applicazioni richiedenti (che d'ora in poi chiameremo genericamente *task*).

Le applicazioni real-time sono caratterizzate da vincoli temporali, i quali possono essere descritti meglio considerando un task τ_i come un flusso di istanze (chiamate *job*) indicate con J_i^k. Diciamo che un job arriva quando un task si sblocca (i.e. quando diventa pronto per l'esecuzione), e termina quando il task si blocca (perché deve aspettare qualche evento particolare prima di diventare nuovamente pronto per l'esecuzione). Ciascun job J_i^k è caratterizzato dai seguenti parametri(vedi Figura 1.1):

- a_i^k: arrival time (istante in cui il job arriva)
- s_i^k: start time (istante in cui il job va in esecuzione per la prima volta)
- f_i^k: finishing time (istante in cui il job termina)
- d_i^k: deadline (tempo massimo entro cui il job deve terminare la propria esecuzione)
- c_i^k: computation time (tempo massimo necessario al processore per eseguire completamente il job senza interruzioni)

Si dice che la deadline è rispettata se $f_i^k \leq d_i^k$. Il computation time di un task nel worst case si indica con $C_i = \max_k c_i^k$.

L'insieme di regole che determinano la scelta del task da mandare in esecuzione tra i task pronti si chiama *algoritmo di scheduling*. Una schedulazione si dice *fattibile* se esiste un assegnamento dei task al processore tale che tutti i task siano completati rispettando un insieme di vincoli prefissati. Un insieme di task τ si dice *schedulabile* se per esso esiste una schedulazione fattibile.

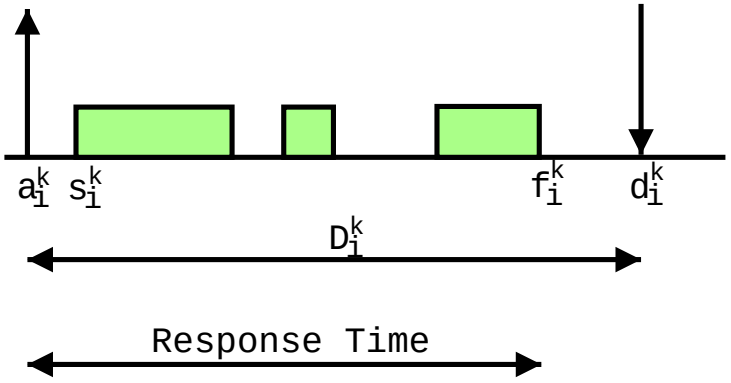

Figura 1.1: Parametri di un job

In un sistema real-time possono esistere i seguenti tipi di requisiti, o vincoli:

Temporali Negli anni recenti abbiamo assistito ad una crescita esplosiva dell'interesse nel supportare applicazioni multimediali come ad esempio applicazioni di streaming audio, e di video conferenza, su sistemi operativi general purpose. Queste applicazioni multimediali, ed in generale tutte le applicazioni soft real-time, sono caratterizzate da vincoli temporali impliciti, che devono essere soddisfatti per fornire la qualità del servizio (*Quality of Service*) desiderata.

Un tipico vincolo temporale è la deadline; altri vincoli temporali possono essere il jitter sulla fine del job, il jitter sullo start time, l'offset iniziale, etc. La deadline è l'unico vincolo temporale che considereremo nella progettazione del nostro sistema operativo real-time.

Le caratteristiche temporali di un task real-time sono anche espresse dalla regolarità delle sue attivazioni (ovvero dal tempo di arrivo dei suoi job), che permette di distinguere tra task *periodici* e *aperiodici*.

Nel caso in cui il task sia periodico, i parametri di ogni suo job (deadline, tempo di arrivo e di computazione) sono costanti. Infatti, un task di questo tipo è caratterizzato dal parametro *periodo* T_i tale che

$$T_i = a_i^{k+1} - a_i^k \qquad \forall k \geq 0$$

e la deadline, se non specificato diversamente, coincide con la fine del periodo T_i.

Di precedenza Nel sistema è presente questo vincolo quando deve essere rispettato un ordine parziale di esecuzione dei task. Generalmente questo vincolo viene descritto mediante un *grafo di precedenza*, che è un grafo *diretto aciclico*.

Su risorse condivise Se un task viene interrotto mentre sta usando una risorsa (ad esempio un dispositivo di I/O, una struttura dati, un database, etc.), la risorsa può rimanere in uno stato inconsistente. Per risolvere questo problema è necessario che la risorsa in questione sia acceduta in *mutua esclusione*, in modo da evitare che più task la utilizzino contemporaneamente.

Una porzione di codice mutuamente esclusiva (cioè non eseguibile da più task contemporaneamente) prende il nome di *sezione critica*. Per implementare la mutua esclusione è possibile usare le *primitive semaforiche*(wait e signal). Nei sistemi real-time, però, il meccanismo semaforico crea problemi, perché prima di mandare un task in esecuzione devo conoscere per quanto tempo occuperà il processore. In particolare l'uso dei semafori è soggetto al fenomeno di *inversione di priorità*, che si verifica quando un task ad alta priorità è bloccato da un task a priorità più bassa per un intervallo di tempo indefinito. Questo fenomeno è indesiderato nei sistemi real-time, poiché introduce ritardi indeterminati sull'esecuzione dei task, con la possibilità di violazione delle deadline critiche (deadline miss).

Il caso classico è quello in cui un task a bassa priorità ha allocato una risorsa di cui ha bisogno un task a priorità maggiore, ma non può rilasciarla perché un task a priorità intermedia ha fatto preemption nei suoi confronti. In questo caso, il task ad alta priorità è costretto ad aspettare sia il task a priorità intermedia, che quello a bassa priorità.

L'inversione di priorità è uno dei problemi più critici nello sviluppo del software per sistemi real-time.

La causa principale dell'inversione di priorità è un interazione nascosta, o progettata male, tra task. L'esempio più lampante di inversione di priorità è il disastro della missione su Marte del Pathfinder: all'interno del sistema operativo (VxWorks) era nascosto un semaforo per la protezione di una `pipe`, condiviso tra un task a bassa priorità ed uno ad alta priorità; il task ad alta priorità subiva una deadline miss ogni volta che un task intermedio faceva preemption sul task a bassa priorità.

In un sistema real-time può essere richiesto che i task abbiano la possibilità di accedere concorrentemente ad una risorsa condivisa, senza che venga violata la consistenza della risorsa stessa.

1.3 Ottimalità

Nel cercare di stabilire quale algoritmo sia migliore di un altro, risulta molto utile la seguente definizione.

Definizione Nel senso della schedulabilità, un algoritmo A^* si dice *ottimo* se gode della seguente proprietà: ' se un insieme di task τ è schedulabile con un generico algoritmo A, allora sicuramente τ sarà schedulabile anche con l'algoritmo ottimo A^*. Viceversa, se τ non è schedulabile con l'algoritmo ottimo A^*, allora sicuramente τ non sarà schedulabile con nessun altro algoritmo.

All'interno di un insieme di algoritmi di scheduling dello stesso tipo, conviene quindi utilizzare l'algoritmo che risulta ottimo, a meno che non sia necessario dare maggiore importanza ad altri fattori (come, ad esempio, l'overhead introdotto).

1.4 Task periodici

Come abbiamo già accennato, i task periodici sono un sottoinsieme dei task aperiodici. Più precisamente, per questo tipo di task risulta periodico il tempo di arrivo a_i^k dei propri job, e possono perciò essere modellati attraverso l'ulteriore parametro $T_i = a_i^{k+1} - a_i^k$.

Per lo studio degli algoritmi di scheduling per task periodici risultano estremamente importanti alcuni parametri che verranno ora illustrati.

Si definisce fattore di utilizzazione di un task il rapporto

$$U_i = \frac{C_i}{T_i}$$

dove C_i e T_i sono, rispettivamente, il tempo di computazione nel worst case ed il periodo, del task τ_i. Questo parametro rappresenta la frazione di tempo utilizzata dal processore per eseguire il task τ_i.

Consideriamo un sistema real-time contenente solo task periodici. Definiamo con il termine *fattore di utilizzazione* del sistema il parametro

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

dove n è il numero di task del sistema. Questo valore è un indice del carico del sistema.

Teorema di non schedulabilità Se $U > 1$ allora non esiste una schedulazione fattibile.

Esiste un limite superiore U_{ub} (upper bound) oltre il quale U non può essere aumentato, altrimenti la schedulazione diventa non fattibile. Tale limite dipende sia dal particolare insieme di task da schedulare, sia dall'algoritmo di scheduling utilizzato.

Dato un algoritmo di scheduling su un insieme di task, il processore si dice pienamente utilizzato dall'insieme se la schedulazione è fattibile, ed un

aumento comunque piccolo del tempo di calcolo C_i di qualsiasi task rende la schedulazione non fattibile.

Per un dato algoritmo, il limite superiore minimo U_{lub}(least upper bound) del fattore di utilizzazione U è il minimo tra i fattori di utilizzazione calcolati su tutti gli insiemi di task che utilizzano pienamente il processore.

Il fattore U_{lub} è un *parametro caratteristico dell'algoritmo di scheduling* e rappresenta una misura del carico massimo gestibile da esso, al di sotto del quale la fattibilità di schedulazione è garantita *per ogni insieme di task.*
Dovendo scegliere tra diversi algoritmi di scheduling, è preferibile quello con il valore più alto del limite superiore minimo U_{lub}: infatti tale è l'algoritmo con il più ampio gruppo di insiemi di task sicuramente schedulabili.
Come vedremo, questo parametro ci guiderà nella scelta dell'algoritmo di scheduling da implementare nel nostro sistema.

1.5 Algoritmi di scheduling

Esistono svariati tipi di algoritmi di scheduling real-time. Per capire meglio a quali di essi siamo interessati nel realizzare il nostro scheduler, e verso quale tipo di problemi essi sono rivolti, cerchiamo di farne una suddivisione secondo alcune caratteristiche principali.

Si indica con *preemption* l'interruzione del task in esecuzione per dare la CPU ad un altro task che ne ha un bisogno più urgente.

Un primo fattore che distingue gli algoritmi di scheduling real-time è la presenza della preemption: un algoritmo *preemptive* è un algoritmo in cui il task in esecuzione può essere interrotto in qualunque momento per dare la CPU ad un altro task.

I sistemi real-time si basano generalmente sul concetto di priorità, ampiamente utilizzato anche nei sistemi operativi tradizionali: ad ogni task viene assegnata una priorità, di cui si serve lo scheduler per scegliere il task da mandare in esecuzione.
Un'altra caratteristica che permette di distinguere gli algoritmi real-time (e che è completamente ortogonale rispetto alla precedente) è il tipo di priorità

utilizzata. Si possono così distinguere due differenti categorie di algoritmi di scheduling: quelli a priorità statiche , e quelli a priorità dinamiche.

Un algoritmo si dice a priorità statiche, se il parametro in base al quale viene stabilito il task da eseguire (i.e. la *priorità* del task) è statico: il suo valore è stabilito al momento dell'attivazione del task, e non può essere in alcun modo modificato durante la vita del task.

Un algoritmo a priorità dinamiche, invece, funziona modificando dinamicamente le priorità dei task presenti nel sistema, in accordo ad una regola ben determinata.

Nonostante gli algoritmi a priorità statiche siano più semplici da realizzare, vedremo che riescono a sopportare un carico minore rispetto agli algoritmi a priorità dinamiche.

Infine, un'altra caratteristica che distingue gli algoritmi di scheduling è la capacità di poter gestire anche i task aperiodici (dei quali i task periodici sono un particolare sottoinsieme).

1.5.1 Rate Monotonic

Consideriamo il caso di task periodici (i.e. $a_i^{k+1} - a_i^k = T_i$, dove T_i è il periodo del task), con tempi di esecuzione nel caso peggiore (C_i) noti.
Per la schedulazione possiamo pensare di utilizzare un algoritmo a priorità fissa. In particolare, l'assegnamento delle priorità chiamato Rate Monotonic garantisce che ciascun task rispetti le proprie deadline a patto che venga superato un test di ammissione.

Si può dimostrare ([Liu73]) che l'algoritmo Rate Monotonic è *ottimo fra gli algoritmi a priorità fissa.*

Rate Monotonic assegna a ciascun task una priorità statica nel seguente modo:

$$p_i = \frac{1}{T_i}$$

Teorema ([Bin01]) Sia $\Gamma = \tau_1, ..., \tau_n$ un insieme di n task periodici, dove ogni task τ_i è caratterizzato da un fattore di utilizzazione U_i. Γ è schedulabile

con Rate Monotonic se

$$\prod_{i=1}^{n}(U_i + 1) \leq 2$$

É stato inoltre dimostrato ([Liu73]) che

$$\lim_{n\to\infty} U_{lub} = \ln 2 \cong 0.69$$

1.5.2 Earliest Deadline First

Questo è uno degli algoritmi più usati nei sistemi real-time con priorità dei task dinamiche. Consiste nel selezionare dalla lista dei job pronti quello la cui deadline assoluta è più imminente. Questo algoritmo assegna a ciascun job una priorità dinamica nel seguente modo:

$$p_i = \frac{1}{d_i}$$

É un algoritmo di tipo preemptive, nel senso che se arriva un task con una deadline minore di quella del task in esecuzione, quest'ultimo viene sospeso e la CPU viene assegnata al task con deadline più bassa appena arrivato.

Si può dimostrare che l'algoritmo EDF è *ottimo fra tutti gli algoritmi preemptive*, e che $U_{lub} = 1$ (cioè un insieme di task è schedulabile se e solo se $U \leq 1$).

Poichè l'algoritmo non sfrutta la periodicità dei task, esso si adatta bene anche ai task aperiodici, e può essere utilizzato per realizzare un sistema ibrido che gestisca sia i task periodici che quelli aperiodici.

1.5.3 Confronto tra RM e EDF

Avendo priorità statiche, l'algoritmo Rate Monotonic risulta più semplice da implementare. Tuttavia, nel comparare i due algoritmi, bisogna tener conto anche dei seguenti fattori:

	Rate Monotonic	Earliest Deadline First
U_{lub}	tende a 0.69	1
ottimalità	solo per priorità fisse	sempre
test di ammissione	complesso	semplice
comportamento in sovraccarico	privilegia i task con periodo più breve	imprevedibile

In sostanza, Rate Monotonic è adatto quando la CPU ha carichi bassi e i tempi di calcolo C_i sono costanti.

Nel nostro sistema operativo real-time implementeremo l'algoritmo di scheduling EDF, poiché permette di garantire la schedulabilità per un carico del sistema (U_{lub}) maggiore, e perché non necessita di un test di ammissione complesso (basta controllare il valore di U).

1.5.4 Server aperiodici

Nel realizzare un sistema per la gestione dei task aperiodici, una tecnica consolidata nel tempo è quella di inserire un nuovo task (chiamato *server aperiodico*) con il compito di gestire l'insieme dei task aperiodici.

Anche per la gestione dei task aperiodici si possono prevedere due tipi di server, a seconda che i task periodici ed il server aperiodico siano schedulati mediante un algoritmo a priorità fissa (i.e. Rate Monotonic) o a priorità dinamica (i.e. Earliest Deadline First).

Noi ci soffermeremo soltanto sugli algoritmi per il servizio delle richieste aperiodiche in un contesto dinamico in cui i processi periodici sono schedulati con l'algoritmo EDF. A differenza dei server a priorità statiche, l'uso di algoritmi dinamici permette infatti di raggiungere la piena utilizzazione della CPU e quindi, a parità di carico periodico, consente di gestire le attività aperiodiche con maggiore efficienza.

Gli algoritmi Constant Bandwidth Server e Greedy Reclamation of Unused Bandwidth, discussi nei successivi capitoli, fanno parte di questo insieme di algoritmi.

Capitolo 2

Constant Bandwidth Server

2.1 Introduzione

Uno dei limiti di alcuni server aperiodici a priorità dinamica (ad esempio *Total Bandwidth Server* [Spu94] [Spu96] oppure *Improved Total Bandwidth* [But99]) è che essi si basano sulla conoscenza dei tempi di calcolo C_i delle richieste aperiodiche da servire. In certi casi, però, il tempo di calcolo di un task può essere sconosciuto, oppure estremamente variabile da un'istanza all'altra (si pensi, ad esempio, ad un MPEG player). Alcune particolari applicazioni, come ad esempio quelle che lavorano su stream audio e video, necessitano di un supporto real-time a causa della loro sensibilità al ritardo ed al jitter. D'altra parte, l'uso di un sistema hard real-time per gestire questo tipo di applicazioni può risultare inappropriato per le seguenti ragioni:

- Se un task multimediale gestisce frames compressi, il tempo per codificare/decodificare ciascun frame può variare significativamente da un'istanza all'altra, perciò il tempo di esecuzione del job nel worst case può essere molto maggiore del suo tempo di esecuzione medio. Poichè le garanzie per i task hard real-time vengono date in base al tempo di esecuzione nel worst case (e non in base al tempo di esecuzione medio), ne risulta che le applicazioni di questo tipo possono causare un enorme spreco delle risorse; il sistema, infatti, viene dimensionato secondo il

worst case di tutti i task real-time, e questo porta ad uno sfruttamento molto parziale delle prestazioni da esso offerte.

- Fornire una stima precisa del tempo di esecuzione nel worst case è molto difficile anche per quelle applicazioni che vanno in esecuzione sempre sullo stesso hardware. Questo problema è ancora più critico per le applicazioni multimediali, le quali generalmente possono andare in esecuzione su un elevato numero di piattaforme differenti (si pensi ad un sistema di video conferenza contemporaneamente in esecuzione su svariate workstation).

- Quando vengono ricevuti dati da un dispositivo esterno (ad esempio, una rete di comunicazione), può essere impossibile determinare un minimo tempo di interarrivo tra i job che hanno il compito di processare quei dati (oppure il tempo di interarrivo è troppo basso, il che porta ad uno spreco di risorse).

- I sistemi multimediali avanzati tendono ad essere più dinamici dei sistemi real-time classici, perciò tutte le metodologie di scheduling inventate per i sistemi real-time statici non si adattano bene a questo nuovo tipo di applicazioni.

Inoltre, il fatto che le garanzie real-time dipendano dalla stima del peggiore tempo di esecuzione di ogni job, rende il sistema fragile rispetto agli errori in questa stima. Se un job non rispetta il tempo di esecuzione stimato, un altro task può violare la propria deadline; in altre parole *non c'è una protezione temporale tra i task nel sistema.*

Naturalmente questo non rappresenta un problema in un sistema dedicato critico: in un tale sistema, infatti, viene tenuto conto di tutti i task già in fase di progettazione; se la stima del worst case del tempo di esecuzione di un task è errata, allora è necessario correggere il progetto dell'intero sistema.

In un sistema operativo general purpose, invece, i task vengono attivati dinamicamente e non è possibile tenerne conto al momento della progettazione. Un task real-time che si comporta male può mettere in serio pericolo la schedulabilità degli altri task (ad esempio un utente può influenzare la

Quality of Service percepita dagli altri utenti), e può mandare in starvation tutti i task del sistema (Denial of Service). Se, poi, il corretto funzionamento del sistema operativo è affidato alla periodica esecuzione di alcuni suoi task, c'è addirittura il rischio di uno stallo dell'intero sistema.

Per questo motivo serve un algoritmo che non si basi sulla conoscenza del tempo di calcolo delle richieste aperiodiche. É inoltre importante che la politica di scheduling possegga le seguenti caratteristiche:

1. Ad ogni task viene garantito un certo livello di servizio
2. Deve esistere un'effettiva protezione tra i task (un task non deve poter causare un degrado nelle prestazioni di altri task).

Un approccio molto utilizzato per modellare il comportamento di un sistema di questo tipo, è quello di associare un server ad ogni task, dove ogni server è caratterizzato da alcuni parametri che specificano esattamente il servizio che egli si aspetta di ricevere dal sistema.

2.2 CBS

L'algoritmo *Constant Bandwidth Server* ([Abe98]) è un *server aperiodico a priorità dinamica.*

I meccanismi che hanno ispirato questo algoritmo sono il *Dinamic Sporadic Server* (DSS) [Spu94] [Gha95] ed il *Total Bandwidth Server* (TBS) [Spu94] [Spu96]. Come il DSS, l'algoritmo CBS garantisce che, se U_s è la frazione del tempo del processore assegnata al server (i.e. la sua banda), il suo contributo al fattore di utilizzazione totale non è maggiore di U_s, anche in presenza di sovraccarichi. Si noti che questa proprietà non è valida per il TBS, per il quale il contributo è limitato a U_s solo sotto l'ipotesi che tutti i job serviti non eseguano per un tempo superiore a quello stimato nel worst case. Rispetto a DSS, tuttavia, CBS ha prestazioni molto più elevate, comparabili con quelle che si possono ottenere soltanto con TBS.

Il CBS può essere definito nel modo seguente:

- Un server è caratterizzato da una capacità C_i e da una coppia ordinata (Q_i,T_i), dove Q_i è il massimo budget e T_i è il periodo del server. Il rapporto $U_i = \frac{Q_i}{T_i}$ è detto *banda* del server. In ogni istante, al server è assegnata una deadline d_i (inizializzata a zero).
  Ogni server mantiene, quindi, le seguenti variabili:

  - C_i : capacità attuale (inizializzata a zero, e che assume valori compresi tra zero e Q_i)
  - d_i: deadline attuale (inizializzata a zero)

- Quando la capacità C_i del server S_i è uguale a zero, essa viene immediatamente ricaricata al suo valore massimo Q_i, e la deadline viene aggiornata con il nuovo valore $d_i = d_i + T_i$. Si noti che non esistono intervalli di tempo finiti in cui la capacità del server è uguale a zero.

- Quando un job J_i^k del server S_i va in esecuzione, gli viene assegnata una deadline uguale alla deadline corrente del server (d_i). Quando il job esegue per un certo intervallo di tempo, la capacità C_i viene decrementata di una quantità pari all'intervallo stesso.

- Un server S_i è detto *attivo* ad un istante t se ci sono job pendenti (si ricordi che la capacità C_i è sempre maggiore di zero), cioè se esiste un job J_i^k tale che $a_i^k \leq t < f_i^k$.
  Un server è detto *idle* al tempo t se in quell'istante non è attivo.

- Quando arriva un job J_i^k ed il server S_i è attivo, la richiesta è accodata in una coda di job pendenti in accordo ad un dato algoritmo (arbitrario) come ad esempio quello FIFO (First In First Out).

- Quando arriva un job J_i^k ed il server S_i è idle, se $C_i \geq (d_i - a_i^k)U_i$ il server aggiorna la propria deadline come $d_i = a_i^k + T_i$, e la capacità C_i viene ricaricata al valore massimo Q_i, altrimenti il job viene servito con i valori correnti della deadline e della capacità del server.

- Quando un job finisce, il successivo job pendente, se esiste, viene servito

con la capacità e la deadline correnti. Se non ci sono job pendenti, allora il server diventa idle.

Riassumendo, ciascun server CBS S_i ha i seguenti parametri (statici):

- T_i: periodo

- Q_i: massima capacità (o budget) che un server aperiodico può consumare in ogni periodo T_i

Gli eventi da servire sono:

- Esaurimento capacità

  Si ricarica immediatamente C_i ($C_i = Q_i$) e si sposta la deadline in avanti ($d_i += T_i$).

- Arrivo di una richiesta quando il server è attivo (i.e. un task del server è in esecuzione)

  La nuova richiesta viene accodata in una coda del server

- Completamento di una richiesta

  Il server preleva dalla coda la successiva richiesta (se esiste) e la schedula con il budget e la deadline attuali.

- Arrivo di una richiesta quando il server è idle (i.e. nessun task del server è in esecuzione)

  Dobbiamo verificare se questa richiesta può essere gestita con i parametri attuali del server (deadline e capacità) oppure è necessario riconfigurare i parametri.
  Indichiamo con t l'istante di arrivo di questa nuova richiesta. Il test da fare è il seguente:
  `if` $(C_i > (d_i - t) * U_i)$ `{`

  $C_i = Q_i;$
  $d_i = t + T_i;$

```
}
```

La schedulazione dei task aperiodici segue le suddette regole, e viene fatta mediante EDF. Quando è in esecuzione un task aperiodico, la capacità C_i del server viene diminuita di Δt ogni Δt di esecuzione.

2.3 Proprietà di CBS

Il meccanismo CBS presenta alcune proprietà interessanti che lo rendono adatto a supportare applicazioni di tipo multimediale. La più importante, la proprietà di *protezione temporale*, è formalmente espressa dal seguente teorema:

Teorema Dato un insieme di n task hard real-time periodici con utilizzazione del processore U_p, ed un CBS con utilizzazione del processore U_s, l'intero insieme è schedulabile con EDF se e solo se

$$U_p + U_s \leq 1$$

L'algoritmo introduce quella che è chiamata protezione temporale tra task: un server aperiodico non occupa la CPU per più di U_i. In questo modo i task periodici e gli altri server aperiodici risultano protetti. La proprietà di protezione temporale ci permette di usare una strategia per riservare la banda, in modo da allocare una frazione del tempo della CPU ai task soft real-time per i quali il tempo di esecuzione non può essere facilmente stimato. La conseguenza più importante di questo risultato è che questi task possono essere schedulati insieme con i task hard real-time senza influenzarne le garanzie fatte a priori, anche nel caso in cui la richieste soft eccedano il carico previsto.

Oltre alla proprietà di protezione temporale, l'algoritmo CBS ha le seguenti caratteristiche.

- Il CBS si comporta come un puro EDF se il task servito τ_j ha i parame-

tri (C_j, T_j) tali che $C_j \leq C_i$ e $T_j = T_i$. Questo è affermato formalmente dal seguente lemma.

Lemma Un task hard real-time τ_j con parametri (C_j, T_j) è schedulabile con un CBS con parametri $C_i \geq C_j$ e $T_i = T_j$ se e solo se τ_j è schedulabile con EDF.

- Il CBS recupera automaticamente ogni tempo avanzato durante le esecuzioni precedenti. Questo è dovuto al fatto che ogni volta che si esaurisce il budget, esso viene immediatamente ricaricato al suo valore massimo e la deadline del server viene postposta. In questo modo il server resta valido ed il budget può essere sfruttato dalle richieste pendenti con la deadline corrente.

- Conoscendo la distribuzione statistica del tempo di esecuzione di un task servito dal CBS, è possibile dare una garanzia statistica, espressa in termini di probabilità di incontrare la propria deadline per ogni job servito.

Sono state fatte diverse simulazioni per confrontare CBS con altri meccanismi simili, cioè *Dinamic Sporadic Server*(DSS) ed il *Total Bandwidth Server*(TBS). La differenza principale tra il DSS ed il CBS è visibile quando il budget è esaurito. Infatti, mentre il DSS diventa idle fino al successivo istante di riempimento (che avviene alla deadline del server), il CBS rimane valido incrementando la sua deadline e ricaricando il budget immediatamente. Questa differenza nell'istante di riempimento determina una grossa differenza nelle prestazioni offerte dai due server nei confronti di task soft real-time.

Anche il TBS non soffre di questo problema, tuttavia il suo corretto comportamento si basa sulla conoscenza esatta del tempo di esecuzione dei job nel worst case, perciò non può essere usato per supportare applicazioni di tipo multimediale.

2.4 Limiti di CBS

Come già detto, l'algoritmo Constant Bandwidth Server è estremamente utile, perché permette di isolare temporalmente i task appartenenti a server diversi, impedendo che il comportamento non corretto di un task porti alla deadline miss di un altro task che si sta comportando correttamente.

Tuttavia, l'algoritmo contiene intrinsecamente alcuni difetti. Consideriamo la schedulazione di Figura 2.1, in cui abbiamo un unico server CBS con budget $Q_i = 2$ e periodo $T_i = 4$ (quindi banda $\frac{Q_i}{T_i} = 0.5$).

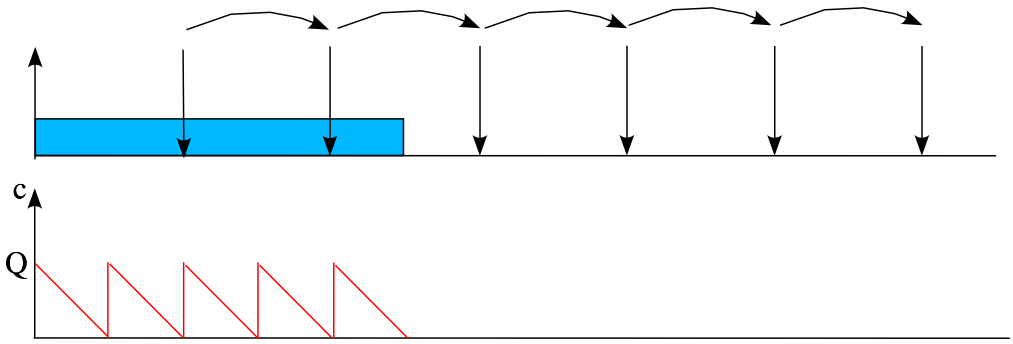

Figura 2.1: Schedulazione con CBS in presenza di un solo task

Si vede chiaramente che ogni due intervalli di tempo la deadline viene postposta di quattro intervalli di tempo. Nonostante il task aperiodico sia l'unico task presente nel sistema, il server CBS continua a postporre la deadline ogni volta che avviene un esaurimento di capacità, anche se non c'è un effettivo bisogno. Infatti, ogni intervallo di tempo di durata Q_i, la capacità del server C_i si esaurisce, e la deadline viene postposta di una quantità $T_i > Q_i$. É facile intuire che, in queste condizioni, prima o poi, si verificherà un overflow, ed il sistema smetterà di funzionare. Notiamo che se il server avesse avuto banda unitaria (cioè $Q_i = T_i$), questo problema non si sarebbe verificato. Questo non è l'unico problema che affligge l'algoritmo: poiché il sistema continua a postporre inutilmente la deadline, se ad un certo istante arriva un task su un altro server, il server che ha eseguito finora è costretto ad aspettare per un lungo periodo di tempo (Figura 2.2). Questo secondo

problema è una conseguenza del primo.

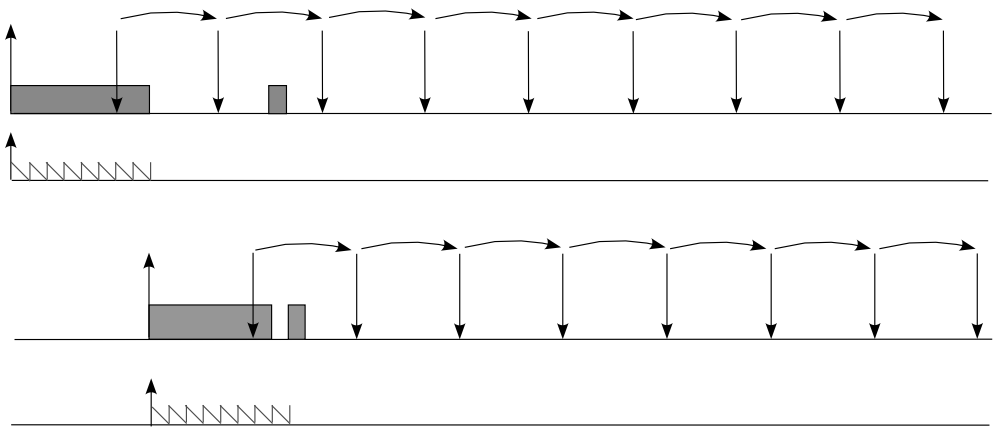

Figura 2.2: Schedulazione con CBS in presenza di due tasks

Consideriamo ora il caso di due server, ciascuno con budget $Q_i = 3$ e periodo $T_i = 10$. La schedulazione che si ottiene con CBS è quella di Figura 2.3.

Come si può vedere, non c'è alcun bisogno che la deadline dei server sia spostata di un periodo $T_i = 10$: sarebbe sufficiente spostarla di $Q_1 + Q_2 = 6$. Perciò il problema dell'algoritmo CBS riguarda *quanto viene spostata la deadline rispetto al valore dei budget*. Anche in questo caso notiamo che se la banda totale del sistema fosse stata unitaria, non avremmo avuto un eccesso nello spostamento della deadline.

Come vedremo nel successivo capitolo, questi difetti dell'algoritmo CBS possono essere risolti mediante l'algoritmo GRUB.

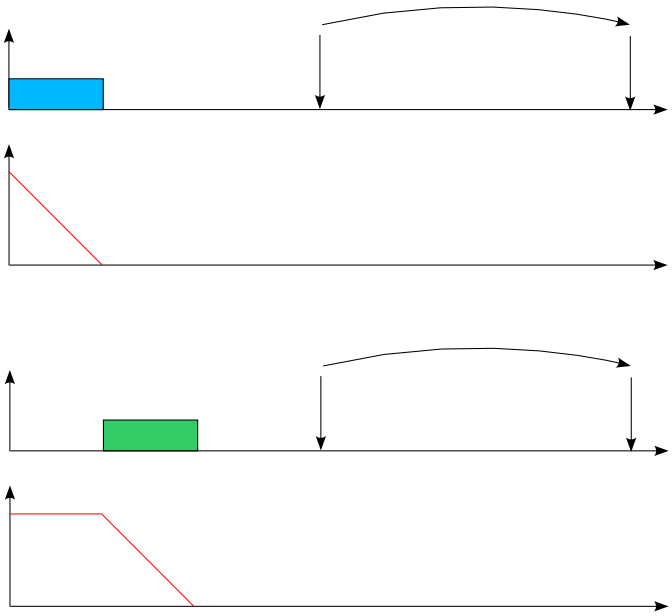

Figura 2.3: Schedulazione con CBS in presenza di due tasks

Capitolo 3

GRUB

3.1 Introduzione

In questo capitolo presentiamo l'algoritmo *Greedy Reclamation of Unused Bandwidth* ([Lip00]), che può essere pensato come una evoluzione dell'algoritmo CBS. Anche GRUB, infatti, riserva una parte della banda del processore per ogni server, e perciò permette una protezione temporale tra i task del sistema.
Rispetto al CBS, GRUB permette di *recuperare la banda inutilizzata del processore*, che non è usata perché alcuni server non hanno job pendenti in attesa di andare in esecuzione. In questo modo si hanno meno spostamenti della deadline, e si risolvono i problemi discussi nel precedente capitolo.

3.2 GRUB

In questo modello ogni server S_i ha i seguenti parametri:

- U_i: banda

- P_i: periodo

La banda U_i rappresenta la parte di capacità totale del processore che deve essere dedicata al task modellato dal server S_i. In parole povere, al server S_i sembra che i propri job siano in esecuzione su un processore virtuale dedicato

con velocità pari a U_i volte la velocità del processore attuale. P_i indica la *granularità* del tempo dal punto di vista del server S_i; più piccolo è il valore di P_i, più il concetto di real-time per il server S_i è a grana fine.

Ogni task τ_i in esecuzione sul server S_i genera una sequenza di job J_i^1, J_i^2, J_i^3, ..., dove il job J_i^j diventa pronto per l'esecuzione ('arriva') al tempo a_i^j ($a_i^j \leq a_i^{j+1} \forall i, j$), ed ha una richiesta di esecuzione uguale a c_i^j unità di tempo.

All'interno di ciascun server, assumiamo che questi job siano eseguiti nell'ordine FIFO, cioè J_i^j deve terminare prima che J_i^{j+1} possa cominciare l'esecuzione.

Vogliamo che la nostra politica di scheduling abbia i seguenti requisiti:

- I tempi di arrivo dei job (a_i^j) non sono noti a priori, ma sono rivelati soltanto a run-time.

- Nemmeno le richieste di esecuzione c_i^j sono note in anticipo: esse possono essere determinate soltanto eseguendo J_i^j fino al completamento.

- L'algoritmo di scheduling si deve attenere alla metodologia di schedulazione real-time tradizionale. In particolare, nel caso peggiore la strategia di scheduling deve essere una piccola variante del noto algoritmo Earliest Deadline First.

Consideriamo un sistema costituito da n server S_1,S_2,...,S_n, dove ogni server è caratterizzato dai parametri U_i e P_i descritti precedentemente. Inoltre, focalizziamo la nostra attenzione sui sistemi dove tutti questi server eseguono su un unico processore condiviso (senza perdita di generalità, si assume che questo processore abbia capacità di calcolo unitaria) . Richiediamo perciò che la somma delle bande di tutti i server non sia maggiore di uno (i.e. $\sum_{i=1}^{n} U_i \leq 1$).

Il seguente teorema stabilisce formalmente le garanzie di performance che possono essere fatte dall'algoritmo GRUB rispetto al comportamento che ciascun server avrebbe se fosse in esecuzione su un processore dedicato. La dimostrazione del teorema verrà data in seguito, dopo aver esposto dettagliatamente l'algoritmo.

Teorema Supponiamo che il job J_i^k voglia cominciare l'esecuzione all'istante di tempo A_i^k, e che tutti i job del server S_i siano eseguiti su un processore dedicato con capacità U_i. Su questo tipo di processore dedicato, J_i^k terminerebbe all'istante di tempo $F_i^k = A_i^k + (c_i^k/U_i)$, dove c_i^k denota il tempo di esecuzione di J_i^k. Se, quando viene usato il nostro scheduler globale, J_i^k termina l'esecuzione all'istante di tempo f_i^k, allora è garantito che

$$f_i^k \leq A_i^k + \lceil \frac{(c_i^k/U_i)}{P_i} \rceil P_i$$

Dalla disuguaglianza precedente, segue direttamente che $f_i^k < F_i^k + P_i$. Questo è ciò che intendiamo quando ci riferiamo al periodo P_i di un server S_i come misura della granularità del tempo dal punto di vista del server S_i: con l'algoritmo GRUB, i job di S_i terminano entro un margine P_i del tempo in cui terminerebbero su un processore dedicato.

Molti altri algoritmi di scheduling basati su server (ad esempio CBS), possono offrire garanzie di performance in qualche modo simili a quella garantita dall'algoritmo GRUB. Tuttavia, l 'algoritmo GRUB possiede un'ulteriore caratteristica che non può essere trovata in molti altri algoritmi: l'abilità di recuperare la capacità inutilizzata del processore ('banda') che non è stata usata perché qualche server non aveva job pendenti in attesa di eseguire. Mentre un recupero di questo tipo non influenza direttamente la garanzia di performance che può essere fatta dall'algoritmo (dato che nel peggiore dei casi potrebbero non esserci task in stato idle, e quindi nemmeno una capacità in eccesso da poter recuperare), esso risulta in un miglioramento delle prestazioni del sistema, in particolare riguardo al numero totale di preemption nello scheduler.

Questo algoritmo differisce dall'approccio del CBS principalmente in due modi. Innanzitutto, è possibile caratterizzare il comportamento dei server in modo più accurato, confrontando la prestazione del server S_i sotto l'algoritmo GRUB con quella che avremmo avuto se fosse stato eseguito su un server dedicato con capacità U_i.

Secondo e più importante, l'algoritmo GRUB è capace di recuperare ef-

ficientemente la capacità in eccesso del processore. Anche CBS recupera la capacità in eccesso del processore, nel senso che al processore non è permesso diventare idle mentre ci sono job che aspettano l'esecuzione; tuttavia l'algoritmo GRUB è capace di recuperare la capacità in eccesso in modo più intelligente di altri algoritmi, permettendo di ottenere una schedulazione migliore (i.e. meno preemption) di quella ottenibile con CBS e server simili.

Diamo ora una descrizione dettagliata dell'algoritmo. Per ogni server S_i nel sistema l'algoritmo mantiene due variabili: una deadline d_i ed un tempo virtuale (*Virtual Time*) V_i.

- Intuitivamente, il valore di d_i ad ogni istante è una misura della priorità che l'algoritmo GRUB concede al server S_i a quell'istante: l'algoritmo infatti esegue essenzialmente una schedulazione EDF basandosi su questi valori di d_i.

- Il valore di V_i ad ogni istante è una misura di quanto servizio riservato al server S_i è stato consumato fino a quell'istante di tempo. L'algoritmo cerca di aggiornare il valore di V_i in modo che, ad ogni istante di tempo, il server S_i abbia ricevuto la stessa quantità di servizio che esso avrebbe ricevuto in un tempo V_i se fosse stato eseguito su un processore dedicato con capacità U_i.

L'algoritmo è responsabile di aggiornare il valore di queste variabili, e fa uso di queste per determinare quale job eseguire ad ogni istante di tempo.

Ad ogni istante di tempo un server S_i può essere in uno dei seguenti stati: `inactive`, `activeContending`, o `activeNonContending`. Lo stato iniziale di ogni server è `inactive`. Intuitivamente al tempo t_0 un server è nello stato `activeContending` se in quell'istante di tempo esso ha alcuni job che aspettano di poter andare in esecuzione; è nello stato `activeNonContending` se ha terminato tutti i job che sono arrivati prima di t_0, ma nel farlo ha esaurito la sua frazione di banda del processore fino a oltre t_0 (i.e. il suo virtual time è maggiore di t_0); è nello stato `inactive` se non ha job che attendono l'esecuzione, e non ha esaurito la sua frazione di banda del processore oltre t_0.

Ad ogni istante di tempo, l'algoritmo GRUB sceglie per l'esecuzione un server che è nello stato `activeContending` (se non ci sono server di questo tipo, allora il processore diventa idle). Tra i server che sono nello stato `activeContending`, l'algoritmo sceglie per l'esecuzione il (successivo job del) server S_i, la cui deadline d_i è la più piccola.

Mentre (un job di) S_i è in esecuzione, il suo virtual time V_i viene incrementato (la velocità esatta di questo incremento verrà specificata dopo); quando S_i non è in esecuzione V_i non cambia. Se ad un qualunque istante questo virtual time diventa uguale alla deadline, allora la deadline viene incrementata di P_i ($d_i += P_i$). Si noti che ciò può determinare il fatto che S_i non sia più il server attivo con deadline minore; in questo caso esso deve lasciare il controllo del processore al server con deadline minore.

Alcuni eventi (interni ed esterni) causano il cambio di stato di un server (si veda il diagramma delle transizioni di stato in Figura 3.1):

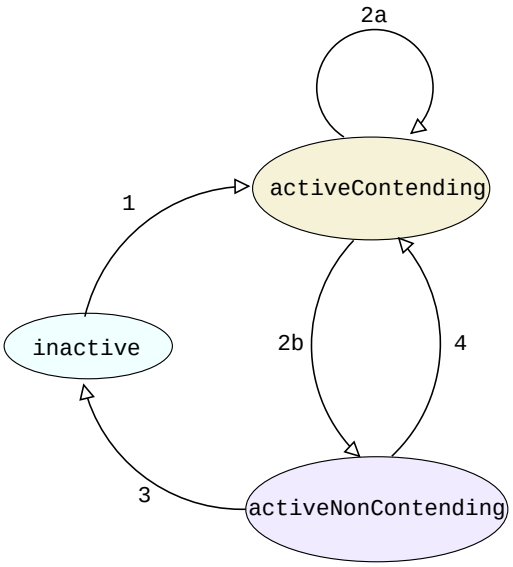

Figura 3.1: Diagramma delle transizioni di stato di GRUB

1. Se un server S_i è nello stato `inactive` e arriva un job J_i^k (ad un istante di tempo a_i^k), allora viene eseguito il seguente codice:

$$V_i = a_i^k$$

$$d_i = V_i + P_i$$

ed il server S_i entra nello stato `activeContending`

2. Quando termina un job J_i^k di un server S_i (si noti che S_i deve essere per forza nello stato `activeContending`), il codice da eseguire dipende dal fatto se il successivo job J_i^{k+1} di S_i è già arrivato oppure no:

   2a. Se è arrivato, allora la deadline è aggiornata nel modo seguente:

   $$d_i = V_i + P_i$$

   ed il server rimane nello stato `activeContending`

   2b. Se non ci sono job di S_i in attesa di eseguire, allora il server S_i può cambiare stato, ed entrare nello stato `activeNonContending`

3. Per un server S_i nello stato `activeNonContending`, è richiesto che ad ogni istante t sia $V_i > t$. Non appena questo cessa di essere vero (il tempo trascorre ma V_i non cambia per i server nello stato `active NonContending`), il server entra nello stato `inactive`. Si noti che il passaggio dallo stato `activeContending` allo stato `inactive` può essere anche immediato, nel caso in cui la relazione $V_i > t$ non sia verificata al momento in cui il server entra nello stato `activeNonContending` (punto 2b).

4. Se arriva un nuovo job J_i^k mentre il server S_i è nello stato `active NonContending`, allora la deadline viene aggiornata nel modo seguente:

   $$d_i = V_i + P_i$$

   ed il server entra nello stato `activeContending`

Resta da specificare come cambia il virtual time V_i di un server S_i, quando è in esecuzione un job di S_i.

Ricordiamo che ciò che ha portato alla progettazione dell'algoritmo GRUB è stata la possibilità di recuperare in modo efficiente la banda del processore

rimasta inutilizzata a causa di alcuni server nello stato `inactive`. Poichè si vuole fare un uso intelligente di questa banda in eccesso, bisogna fare molta attenzione a non esaurire la capacità usando anche la capacità *futura* dei server nello stato `inactive`; infatti non abbiamo idea in quale istante di tempo questi server torneranno ad essere nel loro stato `activeContending`.

L'algoritmo GRUB mantiene la variabile globale *utilizzazione totale del sistema*, che ad ogni istante è uguale a

$$U = \sum_{i=1, S_i \neq inactive}^{n} (\frac{Q_i}{T_i}).$$

dove n è il numero di server presenti nel sistema.

Questa variabile è inizializzata a zero, e viene aggiornata ogni volta che un server entra o esce dallo stato `inactive`. In particolare, quando S_i esce dallo stato `inactive` viene incrementata di U_i, mentre quando S_i entra nello stato `inactive`, essa viene decrementata di un fattore U_i.

Consideriamo un intervallo di tempo $[t, t + \Delta t)$ durante il quale il valore della variabile U non cambia, e durante il quale un job del server S_i è in esecuzione. Vogliamo assegnare al server S_i la capacità in eccesso in questo intervallo. Dato che questa capacità è uguale a $(1-U)\Delta t$, durante il suddetto intervallo di tempo, il server S_i utilizza una quantità di capacità pari a $\Delta t - (1 - U)\Delta t$ cioè $U\Delta t$. Ragionando sul virtual time, anzichè sulla capacità[1], questo è equivalente ad incrementare V_i di una quantità

$$\frac{(U\Delta t)}{U_i}$$

Ne deriva che la regola per l'aggiornamento del virtual time nell'algoritmo GRUB è la seguente:

$$\frac{d}{dt}V_i = \begin{cases} \frac{U}{U_i} & \text{se } S_i \text{ sta eseguendo} \\ 0 & \text{altrimenti} \end{cases}$$

Vediamo infine, come l'algoritmo distribuisce la banda in eccesso tra i

[1]Per una equivalenza dettagliata tra il virtual time di GRUB e la capacità di CBS, vedere il capitolo 6.

server che ne hanno bisogno.

Teorema Se l'utilizzazione del sistema U durante l'esecuzione è superiormente limitata da una costante $c < 1$ (i.e. il valore della variabile U non supera mai c), e se un server S_i ha job da eseguire ad ogni istante (server *backlogged*), allora S_i riceve una frazione di almeno $\frac{U_i}{c}$ della capacità del processore.

Quindi nei sistemi in cui l'utilizzazione del sistema è superiormente limitata da una costante, l'algoritmo alloca la capacità in eccesso ai server che ne hanno bisogno in modo direttamente proporzionale alla loro banda.

Il seguente lemma stabilisce una proprietà dell'algoritmo.

Lemma Ad ogni istante, e durante l'esecuzione di tutti i server S_i, i valori delle variabili V_i e d_i mantenute dall'algoritmo GRUB soddisfano la seguente disuguaglianza:

$$V_i \leq d_i \leq V_i + P_i$$

Consideriamo nuovamente il seguente teorema:

Teorema Sia f_i^k l'istante di tempo in cui l'algoritmo GRUB completa l'esecuzione del job J_i^k. Allora vale la seguente disuguaglianza:

$$f_i^k \leq A_i^k + \lceil \frac{(c_i^k/U_i)}{P_i} \rceil P_i$$

Diamo ora un breve abbozzo della dimostrazione.
Per dimostrare la correttezza del teorema, facciamo le seguenti osservazioni. Per ogni sequenza di arrivo dei job,

- C'è una schedulazione processor-sharing[2] in cui ogni job J_i^k completa

[2]Cioè una schedulazione in cui ad ogni istante di tempo frazioni della capacità del

esattamente o prima dell'istante $A_i^k + \lceil \frac{(c_i^k/U_i)}{P_i} \rceil P_i$. Questa è semplicemente una schedulazione ottenuta schedulando i job di ogni server S_i su un server dedicato di capacità U_i; dato che $\sum U_i \leq 1$, la schedulazione ottenuta sovrapponendo tutte queste schedulazioni singole è la schedulazione processor-sharing desiderata.

- Da questa schedulazione processor-sharing, usando la tecnica di Coffman e Denning ([Cof73]) [3], possiamo ottenere una schedulazione preemptive in cui in ogni istante di tempo esegue al più un job.

- Applichiamo infine il noto risultato riguardante l'ottimalità dell'algoritmo di schedulazione EDF, per asserire che di conseguenza l'algoritmo GRUB genererà una schedulazione in cui ogni job J_i^k completa esattamente o prima dell'istante $A_i^k + \lceil \frac{(c_i^k/U_i)}{P_i} \rceil P_i$.

3.3 Algoritmo

Riassumiamo quanto detto, cercando di focalizzare la nostra attenzione su quali sono i parametri necessari per implementare l'algoritmo.

Per ogni server aperiodico S_i sono necessari i seguenti parametri:

- U_i: banda

- P_i: periodo

Per ogni server S_i sono necessarie, inoltre, le seguenti variabili:

- d_i: deadline (poiché usiamo EDF, indica anche la priorità)

processore possono essere assegnate a vari job differenti, garantendo che la somma di queste frazioni non superi uno.

[3] Per i nostri scopi, questa tecnica si riduce a considerare i massimi intervalli di tempo contigui durante i quali nessun job ha un arrivo o una deadline, e a partizionare questo intervallo tra i job che eseguono al suo interno, in modo che con entrambe le schedulazioni ogni job esegua nell'intervallo per lo stesso ammontare di tempo. Naturalmente, una schedulazione di questo tipo avrebbe un numero di preemption inaccettabile, ma questo non ci riguarda (non stiamo realmente realizzando questa schedulazione, ma ne stiamo soltanto asserendo l'esistenza).

- V_i: virtual time : è una misura della quantità di servizio riservato al server S_i consumata (in ogni istante S_i ha ricevuto lo stesso ammontare di servizio che avrebbe ricevuto in un tempo V_i su un processore dedicato con capacità U_i).

- Stato: all'istante t_0 può essere:

  - `activeContending`: alcuni job di S_i devono terminare e $V_i \leq t_0$
  - `activeNonContending`: i job di S_i arrivati prima di t_0 sono terminati, ma il server S_i ha esaurito la sua banda U_i (cioè $V_i > t_0$).
  - `inactive` (stato iniziale): i job di S_i sono terminati, ed il server S_i non ha ancora esaurito la sua banda U_i.

Il sistema mantiene la variabile globale utilizzazione del sistema U, inizializzata a zero, che ad ogni istante è uguale alla somma delle bande U_i di tutti i server S_i che sono `activeContending` oppure `activeNonContending`. Chiaramente, poiché si utilizza EDF, è necessario che sia $U \leq 1$.

L'algoritmo ha il compito di mantenere aggiornate tutte queste variabili e di mandare in esecuzione ad ogni istante un task scegliendolo tra quelli dei server in stato `activeContending`, secondo l'algoritmo EDF.

Descriviamo il funzionamento dell'algoritmo mediante uno pseudocodice dal significato intuitivo.

Linea	Codice
1	when (V_i == d_i)
2	d_i += P_i;
3	//potrebbe essere necessario un cambio di contesto
4	if ($Stato_i$ == inactive) {
5	V_i = a_i^k;
6	d_i = V_i + P_i;
7	U += U_i;
8	$Stato_i$ = activeContending;
9	//potrebbe essere necessario un cambio di contesto
10	}
11	else if ($Stato_i$ == activeContending)
12	<memorizza a_i^k >
13	else { /* activeNonContending */
14	d_i = V_i + P_i;
15	$Stato_i$ = activeContending;
16	//potrebbe essere necessario un cambio di contesto
17	}
18	if (< J_i^{k+1} è già arrivato >) {
19	d_i = V_i + P_i;
20	//potrebbe essere necessario un cambio di contesto
21	}
22	else if (V_i > t)
23	$Stato_i$ = activeNonContending;
24	else {
25	U -= U_i;
26	$Stato_i$ = inactive;
27	}
28	when (V_i == t && $Stato_i$ == activeNonContending) {
29	U -= U_i;
30	$Stato_i$ = inactive;
31	}

Esecuzione

L'algoritmo esegue sempre il job di un server in stato `activeContending` secondo EDF.
Il virtual time viene aggiornato nel modo seguente:

$$\frac{d}{dt}V_i = \begin{cases} \frac{U}{U_i} & \text{se } S_i \text{ sta eseguendo} \\ 0 & \text{altrimenti} \end{cases}$$

In altre parole, durante l'esecuzione per un tempo Δt, si incrementa il virtual time del server in esecuzione nel modo seguente:

$$V_i \texttt{ += } \frac{U}{U_i}\Delta t\,.$$

Non appena $V_i = d_i$ si incrementa la deadline eseguendo lo pseudocodice compreso nelle linee dalla 1 alla 3.

Arrivo di un job J_i^k

Si esegue lo pseudocodice compreso nelle linee dalla 4 alla 17.

Fine esecuzione del job J_i^k

Si esegue lo pseudocodice compreso nelle linee dalla 18 alla 27.

Transizione da ActiveNonContending a Inactive

Si esegue lo pseudocodice compreso nelle linee dalla 28 alla 31.

Capitolo 4

Hard Reservation

4.1 Descrizione del problema

Come CBS, anche l'algoritmo GRUB, presenta alcune inadeguatezze nel servire determinati tipi di carichi. Consideriamo la schedulazione di Figura 4.1, in cui si hanno due server con periodo molto diverso ($P_2 = 10.5P_1$), entrambi con la stessa banda ($U_1 = U_2$) e con un task backlogged (cioè che ha sempre qualcosa da eseguire).

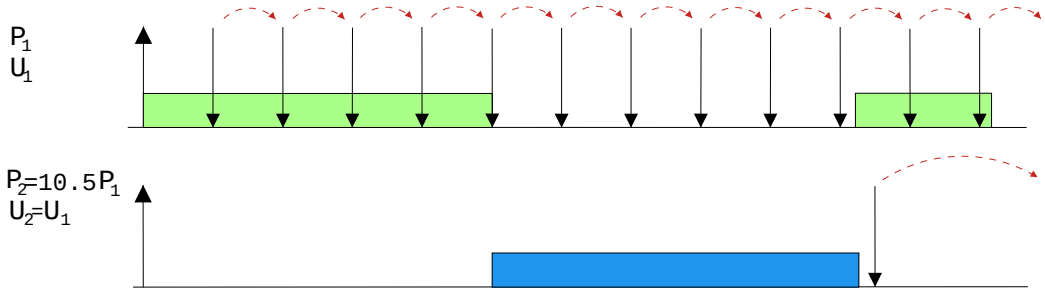

Figura 4.1: Schedulazione con GRUB con due task backlogged

Come si vede, nonostante il primo server abbia un periodo molto breve, cioè una granularità molto fine (segno che ha bisogno di non aspettare troppo tempo tra un'esecuzione e la successiva), è costretto ad aspettare per un periodo di tempo molto lungo a causa dei parametri di un altro server. In altre parole, *i parametri di un server possono influenzare (negativamente)*

l'esecuzione dei task appartenenti ad un altro server.

4.2 Possibile soluzione al problema

Per risolvere questi problemi possiamo pensare ad una piccola modifica da implementare nell'algoritmo, in modo da non avere più questi tempi di attesa indesiderati. La modifica proposta è nota in letteratura con il nome di *Hard Reservation* ([Raj98]).

L'idea è quella di creare un nuovo stato per i server, chiamato *Suspended*, nel quale il server in esecuzione entra al momento in cui postpone la propria deadline in seguito al verificarsi dell' evento $V_i = d_i$. Una volta entrato in questo nuovo stato, il server libera la risorsa CPU, in attesa dell'istante in cui $t = V_i$ (momento in cui il server torna ad essere ActiveContending).

Si fa notare che il comportamento che il sistema deve tenere (con o senza Hard Reservation) dipende dal tipo di task che si sta servendo: in alcuni casi, infatti, la schedulazione senza Hard Reservation risulta più adatta.

Utilizzando nuovamente lo pseudocodice introdotto nel precedente capitolo, possiamo scrivere:

Linea	Codice
32	when (V_i == d_i && $Stato_i$ == ActiveContending) {
33	d_i += P_i;
34	$Stato_i$ = Suspended;
35	//Cambio di contesto
36	}
37	when ($t == V_i$ && $Stato_i$ == Suspended) {
38	$Stato_i$ = ActiveContending;
39	//Potrebbe essere necessario un cambio di contesto
40	}

Il diagramma delle transizioni di stato modificato è mostrato in Figura 4.2.

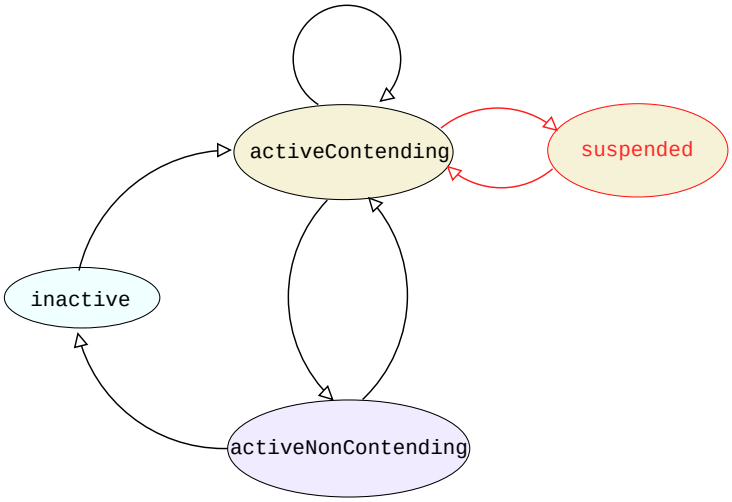

Figura 4.2: Diagramma delle transizioni di stato di GRUB con Hard Reservation

4.3 Correttezza della soluzione

Vogliamo dimostrare che usando l'algoritmo GRUB con Hard Reservation, non si hanno idle times nel caso di task backlogged (cioè task che non si bloccano mai).

Teorema L'algoritmo GRUB con Hard Reservation garantisce che un server S_i con un task backlogged e con periodo P_i esegua esattamente per un tempo $\frac{U_i}{U} P_i$ all'interno del proprio periodo.

Dimostrazione Consideriamo come esempio la schedulazione di Figura 4.3.

All'istante t' il server S_i è pronto per ripartire (perché $t' = V_i$), ma non può ripartire perché non è più il server a priorità maggiore. Affinché il server S_i possa eseguire è necessario che prima i server a priorità maggiore (i.e. S_k e S_h) eseguano abbastanza da spostare la propria deadline.

Generalizziamo il discorso. Consideriamo (vedi Figura 4.4) un generico istante t_1 multiplo di P_i, e l'intervallo di tempo $(t_1, t_1 + P_i)$.

Devo assicurarmi che, anche se tutti gli altri server avessero priorità mag-

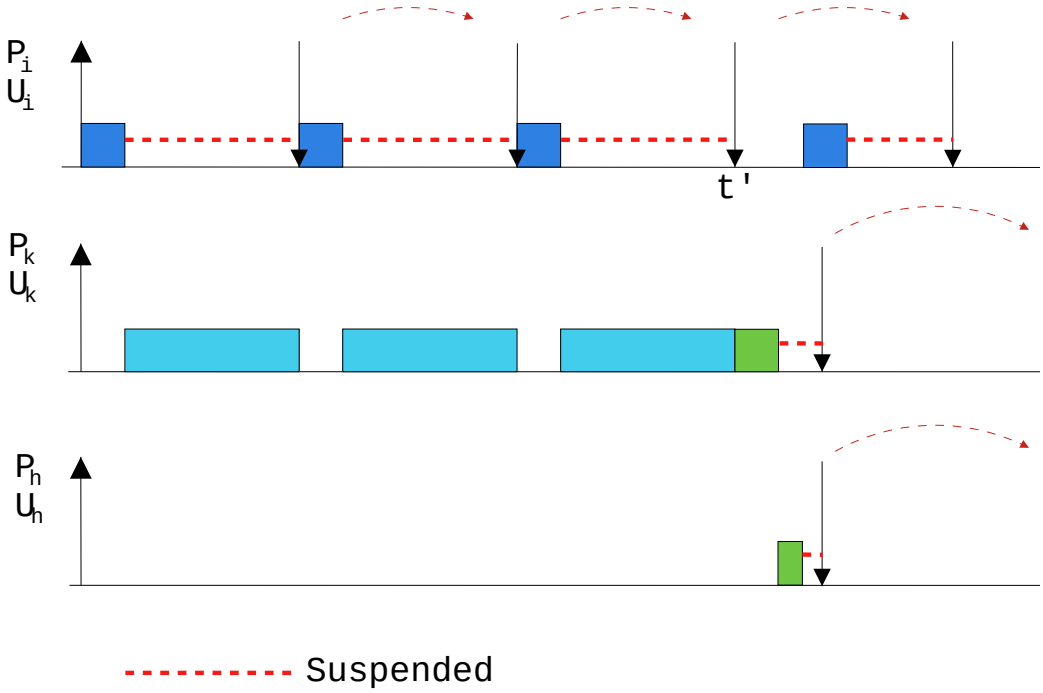

Figura 4.3: Esempio di schedulazione usando GRUB con Hard Reservation

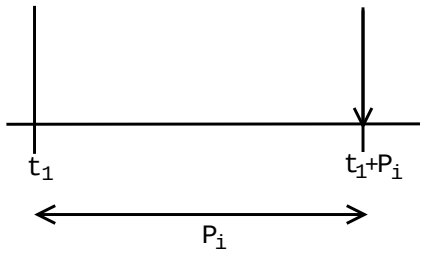

Figura 4.4: Generico periodo P_i

giore del server S_i (cioè la loro deadline fosse minore di $d_i = t_1 + P_i$), il server S_i riuscirebbe comunque ad eseguire per un tempo $\frac{P_i U_i}{U}$. In altre parole, dobbiamo verificare che

$$t_1 + P_i \geq \frac{P_i U_i}{U} + \frac{t_1}{P_i}\frac{P_i U_i}{U} + \sum_{j=1, j\neq i}^{n} \lfloor \frac{t_1 + P_i}{P_j} \rfloor \frac{P_j U_j}{U}$$

$\forall t_1 = kP_i$ con k=0,1,2,...

Vediamo il significato dei termini al secondo membro.

- Il primo termine è quanto deve eseguire il server S_i in questo intervallo.
- Il secondo termine è quanto ha eseguito il server S_i finora.
- Dato che $\frac{P_j U_j}{U}$ rappresenta il tempo di esecuzione necessario a spostare la deadline del server S_j, il terzo termine è il tempo di esecuzione dei server che possono fare (o aver fatto) preemption sul server S_i.

$$t_1 + P_i \geq \frac{P_i U_i}{U} + \frac{t_1 U_i}{U} + \sum_{j=1, j\neq i}^{n} \lfloor \frac{t_1 + P_i}{P_j} \rfloor \frac{P_j U_j}{U}$$

$$t_1 + P_i \geq \frac{U_i}{U}(t_1 + P_i) + \sum_{j=1, j\neq i}^{n} \lfloor \frac{t_1 + P_i}{P_j} \rfloor \frac{P_j U_j}{U}$$

Chiamiamo $t_2 = t_1 + P_i$.

$$t_2 \geq \frac{U_i}{U} t_2 + \sum_{j=1, j\neq i}^{n} \lfloor \frac{t_2}{P_j} \rfloor \frac{P_j U_j}{U}$$

Maggioriamo il secondo termine, sostituendo $\lfloor \frac{t_2}{P_j} \rfloor$ con $\frac{t_2}{P_j}$.

$$t_2 \geq \frac{U_i}{U} t_2 + \sum_{j=1, j\neq i}^{n} \frac{t_2}{P_j} \frac{P_j U_j}{U}$$

$$t_2 - \frac{U_i}{U} t_2 \geq \frac{t_2}{U} \sum_{j=1, j\neq i}^{n} U_j$$

$$\frac{t_2}{U}(U - U_i) \geq \frac{t_2}{U}(\sum_{j=1}^{n} U_j - U_i) = \frac{t_2}{U}(U - U_i)$$

La relazione è verificata.

Perciò il server S_i esegue per almeno $\frac{P_i U_i}{U}$ ogni intervallo P_i. Bisogna inoltre considerare che il server S_i non esegue per un tempo maggiore perché questo tempo di esecuzione è sufficiente a spostare la propria deadline (perciò il server S_i resta nello stato Suspended fino al successivo istante in cui $t = V_i$, cioè all'istante $t_1 + P_i$).

Quindi, all'interno del proprio periodo P_i, il server S_i esegue esattamente per un tempo $\frac{P_i U_i}{U}$.

C.V.D.

Corollario Dato un insieme di server $S = \{S_1, ..., S_n\}$, ciascuno con un task backlogged, usando l'algoritmo GRUB con Hard Reservation si ottiene una schedulazione priva di idle times.

Dimostrazione Consideriamo un periodo di tempo

$$\Delta t = \prod_{k=1}^{n} P_k$$

dove n è il numero di server, e P_k è il periodo del server S_k.

Poichè tutti i server hanno almeno un task backlogged, il precedente teorema afferma che all'interno del periodo Δt ciascun server esegue per un tempo $\frac{U_i}{U}\Delta t$.

Considerando tutti i server, in Δt il processore risulta utilizzato per un tempo pari a

$$\sum_{k=1}^{n} \frac{U_k}{U}\Delta t = \frac{\Delta t}{U}\sum_{k=1}^{n} U_k = \Delta t$$

Perciò non si hanno idle times.

C.V.D.

Corollario Dato un insieme di server $S = \{S_1, ..., S_n\}$, ciascuno con un task backlogged, usando l'algoritmo GRUB con Hard Reservation il tempo di CPU in eccesso viene suddiviso tra i server in modo proporzionale alla loro banda (*fairness*).

Dimostrazione Consideriamo nuovamente un intervallo di tempo

$$\Delta t = \prod_{k=1}^{n} P_k$$

dove n è il numero di server, e P_k è il periodo del server S_k.

Se il generico server S_i eseguisse esattamente con una banda U_i, allora nell'intervallo Δt eseguirebbe per un tempo pari a $U_i \Delta t$.

Considerando gli n server in esecuzione, il tempo di CPU in eccesso da dividere tra essi risulta essere

$$t_{ecc} = \Delta t - \sum_{i=1}^{n} U_i \Delta t = (1 - U)\Delta t$$

Il precedente teorema afferma che all'interno del periodo Δt ciascun server esegue per un tempo $\frac{U_i}{U}\Delta t$.

Perciò il generico server S_i ha eseguito una quantità di tempo in eccesso pari a

$$\frac{U_i}{U}\Delta t - U_i \Delta t = (\frac{U_i}{U} - U_i)\Delta t = \frac{U_i}{U}(1 - U)\Delta t = \frac{U_i}{U} t_{ecc}$$

C.V.D.

Perciò l'algoritmo GRUB con Hard Reservation in presenza di task backlogged è *fair*.

Mostriamo il risultato che abbiamo ottenuto mediante un esempio concreto. Consideriamo la schedulazione di Figura 4.5, in cui $U = U_1 + U_2 + U_3 = 0.8$.

La correttezza di questa schedulazione è avvalorata dalla Figura 4.7 che mostra una schedulazione reale ottenuta implementando GRUB con Hard Reservation su Linux. Come si può vedere, la schedulazione conferma l'assenza di idle times.

Se ogni server avesse eseguito con una banda U_i, allora:

Il server 1 avrebbe eseguito per $P_1 U_1 = 0.32$ in un tempo $P_1 = 2$

Il server 2 avrebbe eseguito per $P_2 U_2 = 4.08$ in un tempo $P_2 = 6.8$

Il server 3 avrebbe eseguito per $P_3 U_3 = 0.28$ in un tempo $P_3 = 7$

Invece, ogni server ha eseguito per $\frac{P_i U_i}{U}$ in un tempo P_i, cioè

Il server 1 ha eseguito per $\frac{P_1 U_1}{U} = 0.4$ in un tempo $P_1 = 2$

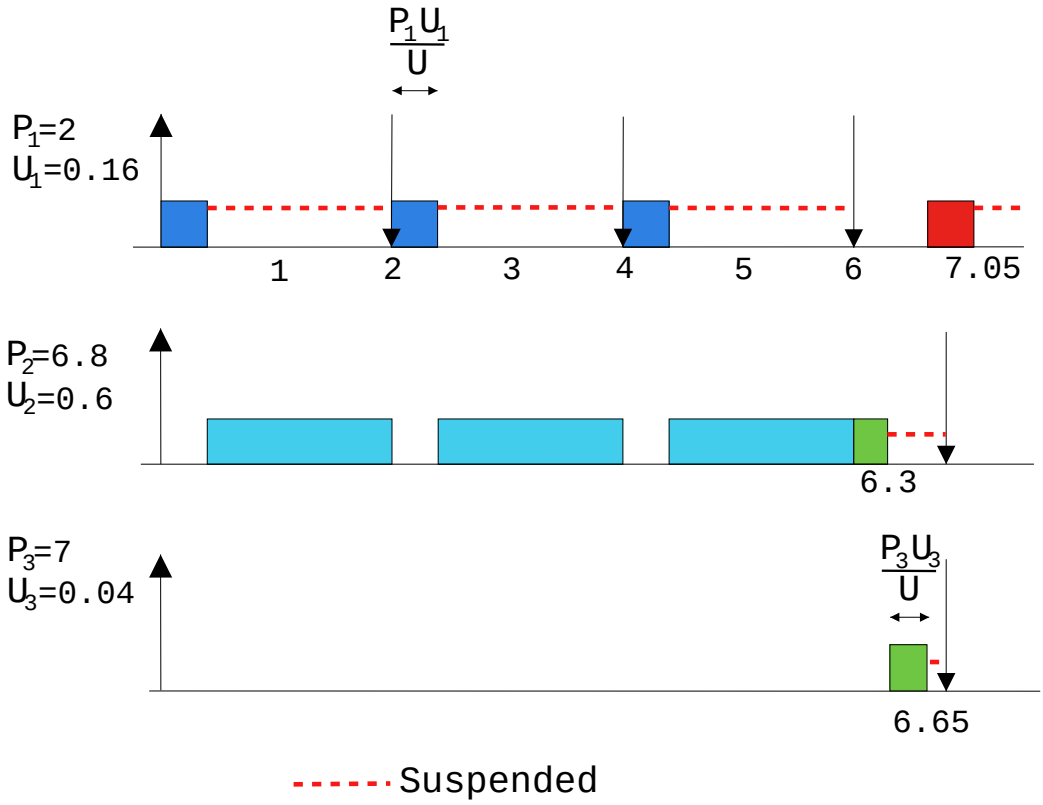

Figura 4.5: Schedulazione con U=0.8

Il server 2 ha eseguito per $\frac{P_2U_2}{U} = 5.1$ in un tempo $P_2 = 6.8$

Il server 3 ha eseguito per $\frac{P_3U_3}{U} = 0.35$ in un tempo $P_3 = 7$

Consideriamo un tempo $\Delta t = P_1P_2P_3 = 95.2$. In questo tempo

Il server 1 ha eseguito in più $\frac{\Delta t}{P_1}(0.4 - 0.32) = 3.808$

Il server 2 ha eseguito in più $\frac{\Delta t}{P_2}(5.1 - 4.08) = 14.28$

Il server 3 ha eseguito in più $\frac{\Delta t}{P_3}(0.35 - 0.28) = 0.952$

Indichiamo con t_{ecc} il tempo di CPU in eccesso, e vediamo come esso è stato suddiviso tra i tre server. Poichè $t_{ecc} = 3.808 + 14.28 + 0.952 = 19.04$, si ha che

$3.808 = t_{ecc}\frac{U_1}{U}$

$$14.28 = t_{ecc}\frac{U_2}{U}$$

$$0.952 = t_{ecc}\frac{U_3}{U}$$

cioè il tempo di CPU in eccesso è stato diviso tra i server in modo proporzionale alla loro banda U_i.

4.4 Hard Reservation in CBS

Facciamo ora alcune considerazioni riguardo l'uso dell'Hard Reservation nell'algoritmo CBS. Proprio a causa dei problemi che l'algoritmo CBS presenta ogni volta che il fattore di utilizzazione totale del sistema è diverso da uno, l'Hard Reservation porta alla *presenza di idle times anche quando tutti i server hanno task backlogged* (si dice che l'algoritmo è *non work conserving*). Si consideri, infatti, il caso di due server con budget $Q_i = 3$ e periodo $T_i = 10$ (si veda la Figura 4.6).

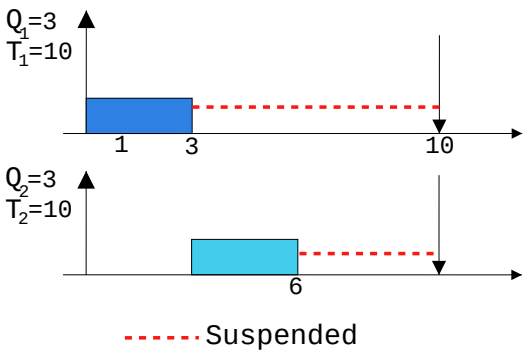

Figura 4.6: Esempio di schedulazione usando CBS con Hard Reservation

In questo caso il primo dei due server che va in esecuzione esegue per tre intervalli temporali, e si sospende per sette intervalli temporali. A questo punto va in esecuzione il secondo server, che esegue per tre intervalli di tempo, e dorme per altri sette intervalli temporali. Dopo sei intervalli di tempo, perciò, il processore diventa idle (per ben quattro intervalli di tempo!) nonostante ci siano due server pronti per l'esecuzione. Come abbiamo già avuto

modo di vedere, questo problema è intrinseco all'algoritmo CBS, e si presenta ogni volta che il fattore di utilizzazione totale è inferiore a uno.
In sostanza, ha senso realizzare l'Hard Reservation nell'algoritmo CBS solo avendo la sicurezza di una utilizzazione del sistema unitaria, oppure nel caso in cui altre considerazioni risultino più importanti della presenza di idle times nella schedulazione (ad esempio quando c'è la necessità di lasciare del tempo libero per eseguire task schedulati in background).

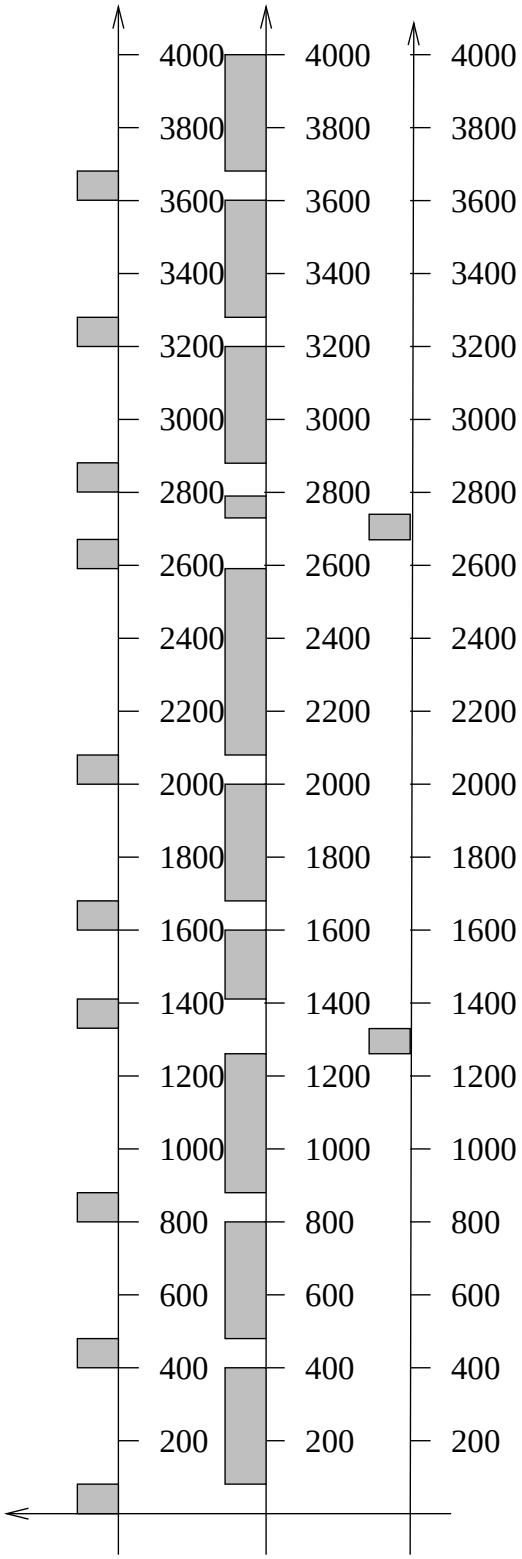

Figura 4.7: Schedulazione di Figura 4.5 in un caso reale

Capitolo 5

Generic Scheduler

5.1 Stato dell'arte

Prima di mostrare la strada seguita nell'implementazione dello scheduler, facciamo uno studio[1] dello stato dell'arte della tecnologia real-time resa disponibile dalla comunità di ricerca, analizzando le caratteristiche di alcuni sistemi operativi real-time general purpose più noti.

Il termine 'sistema operativo real-time' è molto generico, ed indica un ampio insieme di sistemi operativi che forniscono un supporto di qualche tipo per le applicazioni real-time. Questo insieme spazia dal sistema operativo piccolo e sufficientemente semplice da entrare in pochi kbyte di memoria e che può funzionare su processori semplici, ai sistemi operativi che forniscono una completa interfaccia grafica per l'utente ma che richiedono molta memoria ed un processore complesso (con Floating Point, MMU, modalità protetta, etc.).

All'interno della comunità di ricerca, i principali sistemi operativi real-time per macchine general purpose, sono *RTLinux* e *RTAI*. Cerchiamo di descrivere gli aspetti sostanziali di questi due sistemi.

[1] Per uno studio completo vedere [Oce02]

5.1.1 RTLinux

RTLinux ([RTLinux])è una versione di Linux sviluppata da FSMLabs (*Finite State Machine Labs* `http://www.fsmlabs.com/`) in grado di fornire funzionalità hard real-time. Questo sistema operativo è stato impiegato in computer della NASA per raccogliere dati nell'occhio di un famoso ciclone, in campo cinematografico per controllare oggetti animati, e addirittura in importanti esperimenti medici di cardiologia.

Esistono due versioni differenti di RTLinux: una versione sotto licenza GPL[2] (*RTLinux/Open*) ed una versione che non è sotto questa licenza (*RTLinux/Pro*). Dato che il nostro studio è rivolto a soluzioni non commerciali, abbiamo deciso di considerare solo la prima versione, nonostante il suo sviluppo sia stato interrotto nell'anno 2001. Si fa notare che, anche se esiste una versione sotto licenza GPL, l'idea che è alla base di RTLinux è stata brevettata negli Stati Uniti da Victor Yodaiken in data 30 Novembre 1999 (brevetto numero 5995745) [3]. Il detentore del brevetto ha comunque assicurato alla comunità di ricerca che l'uso della versione sotto licenza GPL e lo sviluppo di applicazioni non commerciali non necessiteranno del pagamento dei diritti di autore.

RTLinux è un sistema operativo piccolo e veloce, che segue lo standard POSIX 1003.13 'minimal real-time operating system'. Lo scopo principale nella progettazione di questo sistema (cominciata come progetto di ricerca nel 1995) è quella di trovare una soluzione pacifica ad un apparente contraddizione: la necessità di un sistema operativo hard real-time semplice, e l'importanza di avere molto software complesso (come, ad esempio, software di rete e interfacce grafiche per l'utente).

L'approccio di RTLinux è quello di creare un nuovo abstraction layer al di sotto del kernel di Linux, con il compito di prendere il pieno controllo degli interrupt e del processore. In parole povere, il kernel real-time esegue

[2]La GPL, o *General Public License*, è stata creata dalla comunità di sviluppatori GNU, e permette di modificare codice Open Source a patto di distribuire il codice sorgente modificato. La licenza è disponibile al seguente URL: *http://www.gnu.org/licenses/gpl.html*

[3]La licenza di brevetto può essere consultata all'indirizzo *http://www.fsmlabs.com/products/rtlinuxpro/rtlinux_patent.html*

un sistema operativo general purpose (i.e. Linux) come se fosse un task in background, che va in esecuzione quando non ci sono attività real-time da eseguire. RTLinux crea perciò un layer di hardware virtuale che si frappone tra il kernel standard di Linux e l'hardware vero e proprio del computer. Allo stesso modo di come si comporta il kernel standard di Linux nei confronti dei task applicativi, questo nuovo layer finge di essere l'hardware reale. In questo modo RTLinux implementa un sistema operativo real-time completo e prevedibile, senza alcuna interferenza proveniente dalla parte non real-time di Linux. I thread di RTLinux sono eseguiti direttamente da uno scheduler a priorità fissa. L'intero kernel Linux, e tutti i suoi normali task, sono gestiti dallo scheduler di RTLinux come un task in background.

Ci sono tre modifiche principali da fare al kernel di Linux per virtualizzare l'hardware in modo che RTLinux possa prendere il pieno controllo della macchina:

- Il layer di RTLinux deve assumere il controllo diretto di tutti gli interrupt hardware

- RTLinux deve prendere il controllo del timer hardware ed implementare un timer virtuale per Linux

- Quando Linux ordina all'hardware di disabilitare le interruzioni, il kernel real-time intercetta la richiesta e la registra. In questo modo Linux non può mai essere in grado di disabilitare realmente le interruzioni, ma inconsapevolmente fa una disabilitazione virtuale. Per ottenere questo risultato RTLinux rimpiazza le chiamate alle funzioni `cli` e `sti` (che abilitano e disabilitano le interruzioni).

  Quando arriva un interrupt, il kernel real-time lo intercetta e decide cosa farne. Se esiste un handler per l'interrupt, allora viene invocato. Se non esiste alcun handler, oppure se l'handler indica di voler condividere l'interrupt con Linux, allora l'interrupt viene marcato come pendente. Se Linux ha abilitato gli interrupt, gli interrupt pendenti vengono emulati e vengono invocati gli handler di Linux.

  Non importa cosa faccia Linux, se sia in esecuzione in modalità kernel

o in spazio utente, se abbia disabilitato o no le interruzioni, se sia all'interno di uno spinlock: il sistema real-time è in grado di rispondere all'interrupt in ogni istante.

Ciò che rende RTLinux molto interessante è il fatto che estende l'ambiente di programmazione standard UNIX ai problemi real-time. I thread real-time non possono fare uso dei servizi di Linux perché potrebbero verificarsi inconsistenze o, addirittura , uno stallo. Per superare questo problema, è stato diviso il sistema in due layer: le applicazioni hard vengono eseguite dal layer hard real-time di RTLinux, mentre le applicazioni soft vengono gestite dal layer soft real-time (che viene eseguito come un normale task di Linux). Gli handler real-time degli interrupt possono comunicare con gli ordinari task di Linux attraverso un'interfaccia (dove i task di Linux leggono e scrivono i dati) o attraverso la memoria condivisa.

L'approccio a due layer è una buona soluzione per poter fornire un sistema hard real-time avendo tutte le caratteristiche di un sistema operativo per desktop. Questo approccio separa il meccanismo del kernel real-time dal meccanismo del kernel di Linux general purpose, in modo che ciascuno di essi possa essere ottimizzato in modo indipendente, ed in modo da mantenere il kernel real-time piccolo e semplice.

Le funzionalità offerte dal sistema operativo general purpose, perciò, non sono state duplicate all'interno del kernel real-time: l'ambiente real-time è un ambiente specializzato, e tutto ciò che può essere fatto al di fuori di esso, conviene farlo al di fuori di esso. L'affidabilità, la predicibilità, le prestazioni real-time, e la trasparenza sono rese possibili proprio dal confinare la complessità all'interno del sistema operativo general purpose.

É stato misurato che nel peggiore dei casi, il tempo che trascorre tra quando arriva un interrupt hardware su un processore x86 ed il momento nel quale entra in esecuzione l'handler dell'interrupt, è inferiore a 15 microsecondi (sul kernel standard di Linux sono necessari 600 microsecondi). Sullo stesso hardware, un task periodico real-time è in grado di andare in esecuzione entro 35 microsecondi dall'istante di schedulazione (su Linux servono più di 20 millisecondi!).

RTLinux si affida al meccanismo *Loadable Kernel Module*, offerto da Li-

nux, per installare i componenti del sistema real-time, in modo da mantenere l'intero sistema modulare ed estensibile.

Il sistema è in grado di utilizzare sia uno scheduler basato sull'algoritmo Rate Monotonic, sia uno scheduler basato sull'algoritmo Earliest Deadline First.

5.1.2 RTAI

RTAI (`http://www.aero.polimi.it/~rtai/`) sta per *Real Time Application Interface*. Anche RTAI non è un vero sistema operativo, ma è basato sul kernel di Linux, al quale aggiunge le funzionalità per diventare completamente preemtabile; RTAI è un modulo in stato dormiente, pronto a prendere il controllo su Linux non appena è necessario eseguire un task real-time.

Il progetto è nato come variante di RTLinux presso il dipartimento di Ingegneria Aerospaziale, al Politecnico di Milano, nell'anno 1999. Il progetto è sotto licenza LGPL [4] ed è supportato da una prolifica comunità di sviluppatori basata sul modello Open Source.

Sebbene RTAI contenga ancora codice di RTLinux, le API dei due sistemi (i.e. le interfacce tra il programmatore ed il sistema) si sono evolute seguendo strade molto diverse. Lo sviluppatore (il docente Paolo Mantegazza) ha riarrangiato il codice di RTLinux, aggiungendo nuove funzionalità al sistema, e rendendolo allo stesso tempo più completo e notevolmente più stabile [5]. Rispetto a RTLinux, RTAI può vantare

- un minor numero di bug

- un maggior numero di architetture supportate

- un maggior numero di meccanismi per la comunicazione tra processi

[4] La LGPL, o *Lesser General Public License*, è disponibile al seguente URL: *http://www.gnu.org/licenses/lgpl.html*

[5] La complessità non crea problemi di stabilità o di prestazioni perché il sistema è altamente modulare, e sfrutta la possibilità offerta da Linux di caricare i moduli dinamicamente.

5.1.3 Limiti di RTLinux e RTAI

L'approccio in questi due sistemi operativi real-time è quello di ridurre il kernel Linux ad un semplice task, e lasciare l'intera gestione del sistema al kernel real-time. Per ottenere questo risultato, viene inserito un layer astratto, che finge di essere l'hardware della macchina.

Bisogna notare che questo approccio porta con sè una serie di conseguenze negative:

- Il codice del sistema operativo real-time (i.e. RTLinux o RTAI) è complesso, e dipende fortemente dal codice del kernel standard di Linux. Esiste quindi il potenziale pericolo di dover modificare il codice del sistema real-time ad ogni rilascio di una nuova versione del kernel di Linux, per supplire ai problemi di compatibilità che potrebbero essere sorti. La manutenzione del codice, in queste condizioni, è molto laboriosa.

- Un altro problema è rappresentato dall'impossibilità di usare direttamente i servizi di Linux; nel caso in cui questi servizi possano essere utilizzati, essi eseguono nel layer soft real-time e perciò sono soggetti al rischio di stallo, nel caso in cui i task del layer hard real-time saturino completamente il sistema.

- Entrambi i sistemi operativi real-time diminuiscono l'impredicibilità temporale nel sistema eseguendo Linux come un task in background. Questa soluzione fornisce buone prestazioni per i task real-time, ma non per le applicazioni di Linux, per le quali non è possibile dare alcuna garanzia temporale.

- I processi real-time sono sostanzialmente moduli caricati dinamicamente all'interno del kernel. Questo crea un serio *problema di protezione di memoria*: un task real-time può tranquillamente accedere allo spazio di memoria di un altro task real-time, e ha addirittura il permesso di modificare lo spazio di memoria del kernel! Oltre a rappresentare un problema di sicurezza, ciò rischia di compromettere la stabilità dell'intero sistema.

5.2 Un approccio diverso

A causa di questo gran numero di inconvenienti, pensiamo che l'idea di inserire un layer virtuale all'interno del sistema operativo, non sia la soluzione migliore.

Una possibilità è quella di riscrivere interamente lo scheduler di Linux, ottenendo così un sistema molto efficiente. In quanto a complessità, e a manutenzione, però, questa soluzione risulta ancora peggiore della precedente.

Possiamo allora evitare di riscrivere per intero lo scheduler del sistema operativo, ed applicare al kernel semplicemente una patch (chiamata *Generic Scheduler Patch*) che esporti all'esterno del kernel i simboli necessari alla schedulazione. Una volta disponibili questi dati, Linux offre la possibilità di implementare il nostro scheduler come un modulo da caricare nel kernel al momento opportuno (attraverso il meccanismo *Loadable Kernel Module*).

La struttura del kernel di Linux è simile a quelli dei classici sistemi Unix: usa un'architettura monolitica con file system, driver dei dispositivi, ed altre porzioni di codice collegate staticamente nell'immagine del kernel per poter essere usate subito dopo il boot del sistema. Tuttavia Linux offre un'ulteriore funzionalità: l'uso di moduli dinamici permette di realizzare parti del kernel come oggetti separati che possono essere caricati (comando `insmod`) e scaricati (comando `rmmod`) dinamicamente su un sistema in esecuzione.

Un modulo del kernel è semplicemente un file oggetto contenente funzioni e/o strutture dati da caricare su un kernel in esecuzione. Una volta caricato, il codice del modulo risiede nello spazio di indirizzamento del kernel ed esegue interamente all'interno del contesto del kernel. Più precisamente, un modulo può essere un insieme qualunque di routine, con l'unica restrizione che devono essere fornite anche le funzioni `init_module()` e `cleanup_module()`. La prima funzione è invocata una volta che il modulo è stato inserito; la seconda, invece, prima che il modulo venga scaricato dal kernel.

Una volta che il modulo è stato caricato, esso diventa parte del sistema operativo, perciò può usare tutte le funzioni e accedere a tutte le variabili e strutture dati del kernel. Analogamente, i simboli creati dal nostro modulo vengono resi disponibili agli altri moduli. Se non vogliamo condividere alcuni

simboli (strutture dati o funzioni) basta dichiararli statici.

Il nostro approccio differisce in modo sostanziale dall'approccio usato nei sistemi operativi real-time descritti in precedenza, infatti:

- Non è intrusivo (perciò è più facile la manutenzione del software)
- Si presta particolarmente nel mantenere la compatibilità con lo standard Linux
- Permette di implementare politiche di sicurezza avanzate
- Offre buone prestazioni sia per i task real-time, sia per le normali applicazioni di Linux

Il nostro scheduler deve intercettare l'arrivo dei job (i.e. task che si sbloccano) e la terminazione dei job (i.e. bloccaggio dei task). Inoltre, lo scheduler deve conoscere quando i task vengono creati e distrutti. Per minimizzare la quantità di modifiche necessarie al kernel, possiamo esportare tutti questi eventi rilevanti verso il nostro scheduler, il quale decide quale task schedulare e 'comunica' la decisione allo scheduler del kernel.

In questo modo, lo scheduler può essere implementato come un modulo caricabile che può essere inserito nel kernel a tempo di esecuzione, e la maggior parte del codice dello scheduler risulta indipendente dalla versione del kernel. Perciò è molto semplice portare la nostra implementazione verso nuove versioni di Linux, e usarla in concomitanza con altre patch per il kernel che forniscono caratteristiche utili per applicazioni real-time.

Abbiamo deciso di esportare i servizi per lo scheduling attraverso la system call standard `sched_setscheduler()`, aggiungendo la nuova politica di scheduling `SCHED_CBS` ed estendendo la struttura `sched_param`.

5.2.1 Linux

Per la realizzazione del sistema operativo è stato scelto di usare Linux, data la disponibilità dei sorgenti(fattore indispensabile), ed il gran numero di architetture da esso supportate. Linux è un sistema operativo time-sharing, che offre buone prestazioni e servizi altamente sofisticati. Originariamente Linux

venne progettato per essere usato come server (o, al limite, come desktop). Da allora, Linux si è evoluto ed è cresciuto in modo da poter essere usato in quasi tutti i settori informatici (tra gli altri, i sistemi embedded, i cluster paralleli, e i sistemi real-time).

Il linguaggio di programmazione utilizzato sarà il C, ovvero lo stesso linguaggio utilizzato per scrivere il kernel di Linux e la maggior parte delle sue applicazioni, con piccole porzioni di codice Assembler. Per sviluppare il software è stata usata una distribuzione *Debian* con kernel *2.4.18* (ultima versione stabile disponibile al momento del lancio del progetto OCERA) su una macchina con processore Athlon XP 2000+ (1666 MHz) e 256 Mbyte di RAM.

Purtroppo, l'unico supporto per la schedulazione di attività real-time fornito dal kernel di Linux è dato dall'API di POSIX, la quale fornisce una schedulazione a priorità fissa. POSIX (*Portable Operating System Interface*), è uno standard che è stato sviluppato in maniera congiunta dai gruppi IEEE e The Open Group. Esso definisce interfaccia ed ambiente di un sistema operativo standard (inclusi un interprete di comandi ed alcuni programmi di utilità comune) per supportare la portabilità delle applicazioni a livello di codice sorgente. POSIX è un insieme di standard maturi, ben sviluppati e indipendenti seguiti dalla maggior parte dell'industria UNIX.

Sebbene la schedulazione a priorità fissa sia un eccellente soluzione per implementare attività real-time nei sistemi embedded, essa non si presta bene al caso di schedulazioni soft real-time su sistemi operativi general purpose, dove sono molto importanti i problemi di *fairness* e di sicurezza. Infatti, se ad un utente normale (che non ha privilegi di superuser) è concesso accedere allo scheduler a priorità fissa, un semplice attacco di *denial of service* potrebbe essere attivare un task con la priorità più alta, il quale si limita ad entrare in un loop infinito. D'altra parte, se solo ai superuser è permesso accedere ai servizi della schedulazione real-time, diventa molto difficile fornire garanzie soft real-time agli utenti non privilegiati. Inoltre, anche gli utenti fidati potrebbero far entrare il sistema in starvation durante un debugging. Per questi motivi, un sistema operativo real-time di una workstation dovrebbe supportare uno scheduler che fornisca una *protezione temporale*: un task

non dovrebbe essere influenzato dal comportamento degli altri task presenti nel sistema; sarebbe inoltre desiderabile la possibilità di permettere a tutti gli utenti di accedere ai servizi offerti dal kernel real-time, senza che possano mandare il sistema in starvation. La protezione temporale può essere considerata importante quanto la protezione di memoria, che è fornita da quasi tutti i kernel dei sistemi operativi più comuni.

É stato dimostrato che il *Resource Reservation* è un meccanismo efficace per fornire protezione temporale. Il concetto del Resource Reservation non è nuovo, e algoritmi di schedulazione basati su di esso sono già stati implementati in diverse varianti di Linux. Tuttavia, molte delle implementazioni precedenti soffrono di alcuni problemi, che vanno dalle anomalie nella schedulazione causate da attivazioni aperiodiche, alla mancanza di politiche sulla sicurezza. Questi problemi non sono legati a deficienze intrinseche dell'astrazione del Resource Reservation, ma dipendono unicamente dall'algoritmo di schedulazione usato per l'implementazione.

5.2.2 Ridurre la latenza del kernel

Sebbene Linux non sia un sistema real-time, ha alcune caratteristiche, già incluse nel codice sorgente principale oppure distribuite come patch, progettate per fornire funzionalità real-time. Una di queste patch è la *Preemption Patch* (`http://kpreempt.sourceforge.net`), che dà la possibilità di superare alcuni ostacoli presenti nella struttura del kernel, aiutando a trasformare Linux in un sistema operativo real-time.

La sincronizzazione a 'grana grossa' significa che esistono lunghi intervalli nei quali un task ha l'uso esclusivo di alcuni dati. Questo può ritardare l'esecuzione di un task real-time che ha bisogno delle stesse strutture dati. D'altra parte, la sincronizzazione a 'grana fine' implica che il sistema spenda un sacco di tempo facendo lock e unlock delle strutture dati, rallentando così tutti i task del sistema (compresi quelli real-time). Linux utilizza la sincronizzazione a 'grana grossa' intorno ad alcune strutture dati interne, perché sarebbe stupido rallentare l'intero sistema per diminuire gli effetti nel worst case.

Il maggiore handicap nel considerare Linux come un sistema operativo real-time è che il kernel non è preemptabile; cioè, mentre il processore esegue il codice del kernel, nessun altro task, o evento, può fare preemption sull'esecuzione del kernel. Schedulando un task real-time a livello utente, quindi, le prestazioni real-time possono essere influenzate dalle sezioni non preemptabili del kernel. La *latenza del kernel* è una quantità usata per misurare la differenza tra la schedulazione teorica e quella attuale.

La latenza del kernel è definita nel modo seguente: sia τ_i un task real-time che richiede di andare in esecuzione all'istante di tempo t_1, e sia t_2 il tempo al quale τ_i viene effettivamente schedulato; definiamo la latenza del kernel sperimentata da τ_i come $L = t_2 - t_1$.

La maggiore sorgente della latenza del kernel sono le sezioni del kernel non preemptabili, che possono impedire per un lungo periodo (fino a 100ms) che un task ad alta priorità venga schedulato. Ad esempio, se all'istante t_1 gli interrupt sono disabilitati, il task τ_i potrà entrare nella coda 'pronti' soltanto quando gli interrupt saranno riabilitati. Inoltre, anche se τ_i entrasse nella coda pronti al corretto istante t_1, ed avesse la più alta priorità real-time nel sistema, potrebbe ugualmente non essere schedulato se un altro task fosse in esecuzione nel kernel in una sezione non preemptabile. In questo caso, il task τ_i verrà schedulato quando il kernel uscirà dalla propria sezione non preemptabile al tempo t_2.

La lunghezza di una sezione del kernel non preemptabile dipende dalla strategia usata da esso per garantire la consistenza delle sue strutture dati interne, e dall'organizzazione interna del kernel. Il kernel standard di Linux è basato sulla classica struttura monolitica, nella quale la consistenza delle strutture del kernel è imposta permettendo al più un solo flusso di esecuzione all'interno del kernel ad un dato istante. In questo tipo di kernel *la latenza massima è uguale alla lunghezza massima di una system call, più il tempo di esecuzione di tutti gli interrupt che scattano prima che il sistema torni in spazio utente.*

Dato che la latenza massima può arrivare fino ad un massimo di 100ms, sono state proposte varie patch per evitare che i task real-time sperimentino una latenza di queste proporzioni. In particolare, l'approccio usato dalla Pre-

emption Patch rimuove il vincolo di un'unico flusso di esecuzione all'interno del kernel. Così non è più necessario disabilitare la preemption quando un flusso di esecuzione entra nel kernel. Per supportare una completa preemptabilità del kernel, le strutture del kernel devono essere esplicitamente protette usando mutue esclusioni o il meccanismo di spinlock. *In un kernel preemptabile, la latenza massima è determinata dalla massima quantità di tempo per il quale viene tenuto uno spinlock all'interno del kernel.*

Lo scheduler che realizzeremo deve poter funzionare in concomitanza con la Preemption Patch, in modo da ridurre la latenza per i task real-time, e permettere una schedulazione più precisa. Bisogna considerare, inoltre, che da circa un anno questa patch è stata inclusa nella versione di sviluppo del kernel di Linux (a partire dalla versione 2.5.4) e quindi farà sicuramente parte del kernel standard (al più tardi dalla versione 2.6).

5.3 Funzionalità offerte da Linux

Linux è un'implementazione di UNIX piena di funzionalità. Il criterio principale nella progettazione del kernel di Linux è il throughput, mentre il real-time e la predicibilità non sono un elemento fondamentale. Tuttavia, questo sistema operativo fornisce alcune strutture dati ed alcuni meccanismi che risultano veramente utili nello sviluppo di uno scheduler real-time. Diamo quindi una descrizione dettagliata delle funzionalità offerte da Linux che sfrutteremo per realizzare il nostro scheduler real-time.

5.3.1 Il Current Process

Sebbene i moduli del kernel non eseguano sequenzialmente come le applicazioni, molto spesso le operazioni fatte dal kernel sono correlate ad uno specifico task. Il codice del kernel può conoscere il task corrente che lo sta guidando semplicemente accedendo all'elemento globale `current`, un puntatore ad una struttura di tipo `task_struct`, dichiarata in `<include/linux/sched.h>`. Nella versione 2.4 del kernel `current` è dichiarato in `<asm/current.h>`, incluso da `<linux/sched.h>`.

Il puntatore `current` si riferisce quindi al task utente correntemente in esecuzione: durante l'esecuzione di una system call, come ad esempio open o read, il task corrente è quello che ha invocato la chiamata.

5.3.2 Mutua esclusione

Per l'accesso a risorse condivise si utilizzano gli *spinlocks*. Uno spinlock lavora attraverso una variabile condivisa. Una funzione può acquisire il lock settando la variabile ad un determinato valore. Ogni altra funzione che necessiti del lock può farne richiesta e, vedendo che non è disponibile, entrare in un ciclo in attesa che la variabile diventi disponibile. Una funzione che mantiene uno spinlock per troppo tempo può sprecare molto tempo, poiché le altre CPU sono costrette ad aspettare.
Gli spinlock sono rappresentati dal tipo `spinlock_t`, che, insieme alle funzioni per usarlo, è dichiarato in `<asm/spinlock.h>`.

Generalmente uno spinlock è dichiarato ed inizializzato allo stato unlocked con una linea simile alla seguente:

```
spinlock_t mio_spinlock = SPIN_LOCK_UNLOCKED;
```

Esiste un certo numero di funzioni (generalmente macro) che lavorano con gli spinlocks; tuttavia quelle che ci interessano sono soltanto:

```
spin_lock_irqsave(spin_lock_t *lock, unsigned long flags);
```

> Acquisisce il lock, eventualmente aspettando in un loop finché non è disponibile. Inoltre disabilita le interruzioni sul processore locale, e salva lo stato attuale degli interrupt in `flags`.

```
spin_unlock_irqrestore(spin_lock_t *lock,
                       unsigned long flags);
```

5.3.3 Priorità

Sin dalle prime versioni, lo scheduler di Linux è stato compatibile con lo standard real-time POSIX, includendo, tra le varie politiche di scheduling, quella

a priorità statiche (`SCHED_FIFO`). Esso fornisce inoltre le politiche `SCHED_RR` e `SCHED_OTHER`, richieste dallo standard POSIX. Il range di priorità è [0..99].

É stato fatto molto lavoro per migliorare le prestazioni dello scheduler, attraverso una progettazione minuziosa che ha portato ad un nuovo codice dello scheduler, e ad una nuova struttura. Dalla versione del kernel 2.4.19, il vecchio scheduler è stato rimpiazzato con uno scheduler migliore (di complessità O(1)) realizzato da Ingo Molnar. Questo scheduler è capace di gestire un elevato numero di task senza il degrado dovuto all'overhead.

5.3.4 Linked lists

I kernel dei sistemi operativi, come molti altri programmi, hanno spesso la necessità di mantenere liste di strutture dati. Il kernel di Linux non fa eccezione, ed ospita contemporaneamente numerose implementazioni di liste. Per ridurre la quantità di codice duplicato, gli sviluppatori del kernel hanno introdotto (a partire dalla versione 2.1.45) un'implementazione standard di una lista *circolare* collegata in entrambe le direzioni.

Per usare il meccanismo a lista, è necessario includere il file `<linux/list.h>`. Questo file dichiara una semplice struttura di tipo `list_head`:

```
struct list_head {
     struct list_head *next, *prev;
};
```

Le liste usate nel codice reale sono composte da una serie di strutture piuttosto simili, ognuna delle quali descrive un elemento della lista. Per usare il meccanismo delle liste di Linux nel nostro codice dobbiamo semplicemente racchiudere un `list_head` all'interno della struttura che compone la lista. La testa della lista deve essere una struttura `list_head` a parte, e prima di essere utilizzata deve essere inizializzata con la macro `INIT_LIST_HEAD`.

In `<linux/list.h>` sono definite numerose funzioni che lavorano con le liste:

```
list_add(struct list_head *new, struct list_head *head);
```

Questa funzione aggiunge l'elemento `new` immediatamente dopo l'elemento `head` (generalmente all'inizio della lista). Può quindi essere usata per costruire uno stack. Bisogna notare, comunque, che non è necessario che `head` sia la testa nominale della lista: se si passa una struttura `list_head` che è da qualche parte nel mezzo della lista, il nuovo elemento andrà immediatamente dopo di essa. Infatti, dato che le liste di Linux sono circolari, la testa della lista non è diversa da ogni altro elemento.

```
list_add_tail(struct list_head *new,
              struct list_head *head);
```

Aggiunge un nuovo elemento immediatamente prima di `head` (in altre parole, in fondo alla lista). Può quindi essere usato per costruire code FIFO (First In - First Out).

```
list_del(struct list_head *entry);
```

Rimuove l'elemento dalla lista.

```
list_empty(struct list_head *head);
```

Ritorna un valore diverso da zero se la lista è vuota.

Viene inoltre fornita una macro, `list_entry`, che mappa un puntatore a struttura `list_head` in un puntatore alla struttura che lo contiene. Questa macro può essere invocata nel modo seguente:

```
list_entry(struct list_head *ptr, type_of_struct,
           field_name);
```

dove `ptr` è un puntatore alla struttura `list_head` usata, `type_of_struct` è il tipo di struttura contenente `ptr`, e `field_name` è il nome del campo della lista dentro la struttura.

Il file `<linux/list.h>` definisce inoltre una macro `list_for_each` che espande il codice in un ciclo `for`.

5.3.5 Intervalli di tempo nel kernel

Gli interrupt sono eventi asincroni che generalmente vengono innescati dall'hardware esterno; in seguito ad un interrupt, la CPU interrompe la propria attività corrente, ed esegue del particolare codice (*Interrupt Service Routine*, o ISR) per servire l'interrupt. Il meccanismo usato dal kernel per tenere traccia degli intervalli di tempo, è l'interrupt del timer. Gli interrupt del timer sono generati dall'hardware temporale del sistema ad intervalli regolari; questo intervallo è settato dal kernel secondo il valore di `HZ`, il quale è un valore dipendente dall' architettura, definito in `<linux/param.h>`. Le attuali versioni di Linux settano `HZ` a 100 per la maggior parte delle piattaforme, e a 1024 per alcune piattaforme particolari.

Ogni volta che arriva un interrupt dal timer (cioè ad una frequenza pari a `HZ`), il valore della variabile `jiffies` viene incrementato. `jiffies` è inizializzato a zero durante il boot del sistema, ed è perciò il *numero di clock ticks da quando il computer è stato acceso.* É dichiarato in `<linux/sched.h>` come `unsigned long volatile`, e subisce un overflow dopo un lungo periodo (ma su nessuna piattaforma si ha un overflow in un tempo inferiore a 16 mesi).

Se si vuole un sistema con una frequenza di clock degli interrupt differente, si può modificare il valore di `HZ`. Alcune persone che usano Linux per task hard real-time alzano il valore di `HZ`, per avere tempi di risposta migliori; tuttavia per raggiungere il loro scopo pagano l'overhead degli interrupt aggiuntivi del timer. Tutto sommato, il migliore approccio verso gli interrupt temporali, è quello di mantenere inalterato il valore di `HZ`, riponendo una piena fiducia negli sviluppatori del kernel, che certamente hanno scelto il valore migliore.

Il codice del kernel può sempre reperire il tempo corrente guardando il valore di `jiffies`. Questo valore può anche essere usato per calcolare intervalli di tempo tra eventi. Quando è necessario misurare intervalli di tempo, il valore di `jiffies` è quasi sempre sufficiente.

5.3.6 I timer del kernel

La risorsa definitiva per mantenere il tempo nel kernel è il timer. I timer sono utilizzati per schedulare l'esecuzione di una funzione (*handler* del timer) ad un particolare tempo nel futuro. La funzione registrata nel timer viene eseguita una volta sola: *i timer non sono ciclici.*

Un timer è molto semplice da usare: si registra la funzione una volta, ed il kernel provvede a chiamarla una sola volta quando il timer scatta.

I timer del kernel sono organizzati in una lista; ciò significa che possiamo creare quanti timer vogliamo. Un timer è caratterizzato dal suo valore di timeout (espresso in jiffies) e dalla funzione da chiamare quando il timer scatta. L'handler del timer accetta un argomento, che viene memorizzato nella struttura dati, insieme con un puntatore all'handler stesso.

La struttura dati di un timer è simile alla seguente, che è stata estratta da `<linux/timer.h>`:

```
struct timer_list {
     struct list_head list;
     unsigned long expires;   /*  timeout, in jiffies   */
     unsigned long data;      /* argomento dell'handler */
     void (*function)(unsigned long);        /* handler */
};
```

Il timeout del timer è un valore in jiffies. Così `timer->function` partirà quando `jiffies` sarà uguale o maggiore di `timer->expires`. Il timeout è un valore assoluto; generalmente è calcolato prendendo il valore corrente di `jiffies` e aggiungendo il delay desiderato.

Una volta che una struttura `timer_list` è inizializzata, `add_timer()` la inserisce in una lista ordinata, che viene controllata più o meno ogni 10ms. Anche i sistemi che lavorano con una frequenza di clock degli interrupt molto alta non controllano la lista più spesso: la maggiore risoluzione del timer non giustificherebbe il costo degli ulteriori passaggi attraverso la lista.

Per lavorare sui timer sono disponibili le seguenti funzioni:

```
void init_timer(struct timer_list *timer);
```

Questa funzione è usata per inizializzare la struttura del timer. É fortemente raccomandato di usare questa funzione per inizializzare un timer, e non cambiare mai esplicitamente i puntatori nella struttura, in modo da essere compatibili in futuro.

`void add_timer(struct timer_list *timer);`

Questa funzione inserisce il timer nella lista globale dei timer attivi.

`void del_timer(struct timer_list *timer);`

Questa funzione deve essere invocata se è necessario rimuovere un timer dalla lista prima che esso scatti. Quando un timer scatta, d'altra parte, esso viene automaticamente rimosso dalla lista.

`void del_timer_sync(struct timer_list *timer);`

Questa funzione lavora come `del_timer()`, ma inoltre garantisce che, quando ritorna, la funzione del timer non sia in esecuzione su alcuna CPU.

5.4 Uso delle funzionalità di Linux

5.4.1 Hooks

Il nostro scheduler è stato realizzato come un modulo caricabile che, una volta inserito nel kernel, abilita una nuova politica di scheduling, chiamata `CBS_SCHED`.

Un kernel patchato con la Generic Scheduler Patch, e compilato con l'opzione Generic Scheduler abilitata, esporta alcuni simboli che possono essere usati dal nostro scheduler per intercettare gli eventi necessari alla schedulazione. Questi simboli sono: `block_hook`, `unblock_hook`, `fork_hook`, `cleanup_hook`, `setsched_hook`.

Le variabili riferite da questi simboli sono puntatori a funzioni, chiamati *hooks*. Per ogni hook, il kernel patchato contiene un puntatore a funzione che

viene esportato per poter essere usato dal modulo caricabile. Tutti i puntatori sono inizialmente uguagliati a `NULL`. Quando il modulo viene inserito, esso setta questi puntatori uguali ai suoi handlers, in modo che il kernel invochi l'handler adeguato quando si verifica un determinato evento:

block_hook è invocato quando un task si blocca, in modo che lo scheduler capisca che il job corrente ha finito di eseguire.

unblock_hook è invocato quando un task si sblocca, in modo da informare lo scheduler dell'arrivo di un nuovo job.

fork_hook è invocato quando viene creato un nuovo task ed un puntatore al task creato è passato come parametro.

cleanup_hook è invocato quando un task viene distrutto, in modo che lo scheduler possa deallocare le risorse associate ad esso.

setsched_hook è invocato quando sono invocate le system call `sched_setscheduler()` o `sched_setparam()`.

Tutti gli hooks tranne `setschedsched_hook` ricevono come parametro il puntatore alla struttura `task_struct` del task interessato.

Per convenienza, la patch provvede ad inserire un nuovo campo `private_data` nella struttura `task_struct`. Questo campo è un puntatore a `void` che viene usato dallo scheduler per puntare ai dati real-time privati di ogni task (nel caso di CBS e di GRUB esso punta al server che gestisce il task). Se necessario, lo scheduler deve settarlo al valore opportuno durante la `fork_hook`. Quando il modulo viene rimosso, si deve assicurare che tutti i task abbiano `private_data` settato a NULL.

Si noti che in questa implementazione l'algoritmo di scheduling non dipende dalla periodicità dei task: infatti lo scheduler intercetta direttamente le attivazioni/disattivazioni dei task, ed è responsabilità del task implementare un comportamento periodico.

Nel nostro modulo possiamo quindi scrivere il seguente codice:

```
extern void generic_request(struct task_struct *t);
```

```
extern void generic_release(struct task_struct *t);
extern void generic_register(struct task_struct *t);
extern void generic_unregister(struct task_struct *t);
extern int generic_setsched(pid_t pid, int policy,
           struct sched_param *param);

int init_module(void)
{
  if (block_hook != NULL) {
    printk("Error installing generic scheduler:
            block_hook != NULL!!!\n");
    return -1;
  }
  if (unblock_hook != NULL) {
    printk("Error installing generic scheduler:
            unblock_hook != NULL!!!\n");
    return -1;
  }
  if (fork_hook != NULL) {
    printk("Error installing generic scheduler:
            fork_hook != NULL!!!\n");
    return -1;
  }
  if (cleanup_hook != NULL) {
    printk("Error installing generic scheduler:
            cleanup_hook != NULL!!!\n");
    return -1;
  }
  if (setsched_hook != NULL) {
    printk("Error installing generic scheduler:
            setsched_hook != NULL!!!\n");
    return -1;
  }
  block_hook = generic_release;
  unblock_hook = generic_request;
```

```
  fork_hook = generic_register;
  cleanup_hook = generic_unregister;
  setsched_hook = generic_setsched;
  //...
  return 0;
}

void cleanup_module(void)
{
  block_hook = NULL;
  unblock_hook = NULL;
  fork_hook = NULL;
  cleanup_hook = NULL;
  setsched_hook = NULL;
}
```

5.4.2 Priorità

Vogliamo realizzare il nostro scheduler come un modulo caricabile, che si occupi di variare la priorità dei task in esecuzione sul sistema, in modo che lo scheduler del sistema operativo faccia la schedulazione desiderata (in pratica lo scheduler di Linux ha il solo compito di fare il dispatching dei task).

Basandosi sulle informazioni raccolte tramite gli hook, il nostro scheduler decide quale task eseguire e, tramite la funzione `cbs_schedule()`, fa il dispatching del task da schedulare. Dato che abbiamo deciso di non modificare lo scheduler originale di Linux, il modulo forza il dispatching del task settando la sua policy a `SCHED_FIFO` o `SCHED_RR`, e settando il parametro `rt_priority` alla massima priorità real-time di Linux + 1 (100).

All'interno del modulo saranno quindi presenti le seguenti funzioni:

```
void dispatch(struct task_struct *t, int cpu)
{
  if (t != NULL) {
    if((t->policy != SCHED_FIFO) && (t->policy != SCHED_RR)){
      sched_error("Scheduling ERROR: Scheduling task %d
```

```
      with policy %ld != SCHED_FIFO | SCHED_RR...\n",
      t->pid, t->policy);
    }
    t->rt_priority = SCHEDULING_PRIORITY;
  }
  current->need_resched = 1;
}

void idle_cpu(int cpu)
{
  dispatch(NULL, cpu);
}
```

5.4.3 Timer

Possiamo realizzare il nostro timer mediante la seguente struttura:

```
struct t_data_struct {
  struct timer_list timer;
  void (* callback)(unsigned long long int time, void *param);
  void *param;
  unsigned long long int start_time;
  unsigned long long int requested_interval;
#ifdef __CBS_DEBUG__
  int id;
#endif
};
```

Vediamo il significato dei vari campi:

- Il campo `timer` è del tipo `timer_list`, ossia il timer fornito dal sistema operativo.

- Il secondo campo è un puntatore ad una funzione del tipo

  ```
  void func (unsigned long long int time, void *param);
  ```

Questa è la funzione che viene invocata allo scattare del timer, ed ha come parametri il tempo attuale ed il puntatore ad un eventuale parametro.

- Il campo `param` è un puntatore all'eventuale parametro che deve essere passato alla funzione di callback.

- Il quarto ed il quinto parametro sono indicazioni temporali (rispettivamente l'istante in cui è stato setttato il timer ed il delay di tempo desiderato).

- L'ultimo parametro è utilizzato a scopo di debugging, ed è l'identificatore del timer.

La struttura può essere usata mediante le seguenti funzioni:

```
void sched_stop_timer(struct t_data_struct *tdata);
void sched_set_timer(unsigned long long int t,
                     struct t_data_struct *tdata);
void sched_init_timer(struct t_data_struct *tdata);
```

5.4.4 Trattare interi a 64 bit su architetture i386

Quando abbiamo a che fare con variabili temporali (di cui la deadline è un esempio) le variabili intere a 32 bit possono risultare insufficienti. Nasce il problema di poter usare variabili intere a 64 bit anche su sistemi a 32 bit, avendo la possibilità di fare con esse le classiche operazioni di moltiplicazione e di divisione.

Per supplire a questa necessità possiamo utilizzare la funzione `llimd()` fornita dal gruppo di ricerca di RTAI. Questa funzione accetta 3 parametri (rispettivamente a 64, 32 e 32 bit), moltiplica il primo parametro per il secondo, e divide il risultato per il terzo parametro.
La definizione della funzione è la seguente:

```
/* This function comes from RTAI (http://www.rtai.org) */
static inline long long llimd(long long ll, int mult, int div)
{
```

```
  __asm__ __volatile (\
    "movl %%edx,%%ecx; mull %%esi;       movl %%eax,%%ebx;  \n\t"
    "movl %%ecx,%%eax; movl %%edx,%%ecx; mull %%esi;        \n\t"
    "addl %%ecx,%%eax; adcl $0,%%edx;    divl %%edi;        \n\t"
    "movl %%eax,%%ecx; movl %%ebx,%%eax; divl %%edi;        \n\t"
    "sal $1,%%edx;     cmpl %%edx,%%edi; movl %%ecx,%%edx;  \n\t"
    "jge 1f;           addl $1,%%eax;    adcl $0,%%edx;     1:"
    : "=A" (ll)
    : "A" (ll), "S" (mult), "D" (div)
    : "%ebx", "%ecx");
  return ll;
}
```

5.4.5 Il tempo nell'architettura i386

Nelle architetture `i386`, il TSC (*Time Stamp Counter*) è un registro interno alla CPU che viene incrementato ad ogni ciclo di clock. La funzione `rdtsc` legge il TSC e lo mette nei registri EAX e EDX (infatti TSC è un registro a 64 bit). Il valore del registro può essere letto mediante la seguente funzione:

```
static inline unsigned long long sched_read_clock(void)
{
  unsigned long long res;
  __asm__ __volatile__( "rdtsc" : "=A" (res));
  return res;
}
```

Per poter passare dal valore del TSC (che dipende dalla frequenza di clock della CPU) al relativo valore in microsecondi, si usano le seguenti variabili, definite in `<arch/i386/kernel/time.c>`:

```
unsigned long cpu_khz;  /* Detected as we calibrate the TSC */

/* Cached *multiplier* to convert TSC counts to microseconds.
 * (see the equation below).
 * Equal to 2^32 * (1 / (clocks per usec) ).
 * Initialized in time_init.
```

```
 */
unsigned long fast_gettimeoffset_quotient;
```

Il valore di queste variabili, insieme al valore della variabile `HZ` permette di passare agevolmente dal valore del tempo secondo l'unità di misura del TSC ('clock'), al valore espresso in microsecondi o in jiffies; le funzioni che permettono di fare queste conversioni sono:

```
/* fast_gettimeoffset_quotient = 2^32 * (1/(TSC clocks per usec)) */
static inline unsigned long long int us2clock(unsigned long u)
{
  return llimd(u, cpu_khz, 1000);
}

static inline unsigned long int clock2ms(unsigned long long c)
{
  return (unsigned long int)
         (llimd(c, fast_gettimeoffset_quotient, 1) >> 32) / 1000;
}

static inline unsigned long int clock2us(unsigned long long c)
{
  return (unsigned long int)
         (llimd(c, fast_gettimeoffset_quotient, 1) >> 32);
}

static inline unsigned long int clock2jiffies(unsigned long long c)
{
  unsigned long long int tmp;
  tmp = llimd(c, fast_gettimeoffset_quotient, 1) >> 32;
  return llimd(tmp, HZ, 1000000);
}
```

Si noti che, piuttosto che fare una divisione, di preferisce fare una moltiplicazione ed uno shift, in modo da ridurre i tempi di calcolo.

5.4.6 Generic Scheduler Patch

Riassumendo, sul kernel di Linux per l'architettura i386, la Generic Scheduler Patch provvede a modificare i seguenti file:

- `Makefile`
  Viene modificata la variabile `EXTRAVERSION` aggiungendogli la stringa `-gensched`.

- `arch/i386/config.in`
  Viene aggiunta la seguente linea

  ```
  bool 'Generic Scheduler' CONFIG_GENERIC_SCHEDULER
  ```

  necessaria per poter settare l'opzione 'Generic Scheduler' tramite il comando `make xconfig`.

- `arch/i386/kernel/i386_ksyms.c`
  Si esportano i valori delle variabili `fast_gettimeoffset_quotient` e `cpu_khz`:

  ```
  extern unsigned long fast_gettimeoffset_quotient;
  extern unsigned long cpu_khz;
  EXPORT_SYMBOL(cpu_khz);
  EXPORT_SYMBOL(fast_gettimeoffset_quotient);
  ```

- `include/linux/sched.h`
  In questo file si aggiunge il puntatore `private_data` alla struttura `task_struct`:

  ```
  void *private_data;
  ```

  e si dichiarano i vari hook:

  ```
  extern void (*block_hook)(struct task_struct *tsk);
  extern void (*unblock_hook)(struct task_struct *tsk);
  extern void (*fork_hook)(struct task_struct *tsk);
  extern void (*cleanup_hook)(struct task_struct *tsk);
  extern int (*setsched_hook)(pid_t pid, int policy,
                              struct sched_param *param);
  ```

- kernel/exit.c
  Si definisce il cleanup_hook:

```
void (*cleanup_hook)(struct task_struct *tsk);
```

  e si inserisce il seguente codice nella funzione do_exit:

```
if (cleanup_hook != NULL) {
   cleanup_hook(tsk);
}
```

- kernel/fork.c
  Si definisce il fork_hook:

```
void (*fork_hook)(struct task_struct *tsk);
```

  e si inserisce il seguente codice nella funzione do_fork:

```
p->private_data = NULL;
if (fork_hook != NULL) {
   fork_hook(p);
}
```

- kernel/ksyms.c
  Si esportano i vari hook:

```
extern struct task_struct * init_tasks[NR_CPUS];
EXPORT_SYMBOL(init_tasks);
EXPORT_SYMBOL(block_hook);
EXPORT_SYMBOL(unblock_hook);
EXPORT_SYMBOL(fork_hook);
EXPORT_SYMBOL(cleanup_hook);
EXPORT_SYMBOL(setsched_hook);
```

- kernel/sched.c
  Si definiscono gli hook block_hook, unblock_hook e setsched_hook:

```
void (*block_hook)(struct task_struct *tsk);
void (*unblock_hook)(struct task_struct *tsk);
int (*setsched_hook)(pid_t pid, int policy,
                     struct sched_param *param);
```

Si inserisce il seguente codice nella funzione `add_to_runqueue`:

```
if ((unblock_hook != NULL) && (p->private_data != NULL)) {
   unblock_hook(p);
}
```

Si inserisce il seguente codice nella funzione `sys_sched_setscheduler`:

```
if (setsched_hook != NULL) {
   int res;
   res = setsched_hook(pid, policy, param);
   if (res <= 0) {
      return res;
   }
}
```

Si inserisce il seguente codice nella funzione `sys_sched_setparam`:

```
if (setsched_hook != NULL) {
   int res;
   res = setsched_hook(pid, -1, param);
   if (res <= 0) {
      return res;
   }
}
```

Capitolo 6

Implementazione degli algoritmi

Nel precedente capitolo abbiamo visto perché conviene utilizzare Linux come sistema operativo di partenza e quali strutture dati e meccanismi esso offre per poter implementare il nostro scheduler real-time. In particolare, è stata sottolineata la convenienza nel realizzare lo scheduler come un modulo del kernel caricabile a run-time, e quindi la necessità di realizzare una patch (chiamata *Generic Scheduler Patch*) per poter esportare al di fuori del kernel i simboli necessari alla schedulazione.

Implementata la suddetta patch, resta da vedere come realizzare il modulo dello scheduler. Anche per il codice dello scheduler faremo un ampio uso dei meccanismi nativi offerti dal sistema operativo, e di tutte le strutture dati (timer, liste, funzioni di conversione) che abbiamo derivato a partire da essi, e la cui implementazione è già stata mostrata nel precedente capitolo.

6.1 Implementare CBS

Nonostante l'obiettivo della tesi sia quello di realizzare une versione leggermente modificata dell'algoritmo di scheduling GRUB, cominciamo con il vedere una possibile implementazione del suo predecessore, l'algoritmo CBS. Questa tesi, infatti, fa parte del progetto OCERA, all'interno del quale è possibile trovare le implementazioni della Generic Scheduler Patch e dell'algoritmo CBS, entrambe curate da Luca Abeni ([Abe02]).

Cominciamo con il vedere come è possibile implementare un server CBS. Nell'ipotesi semplificativa in cui ciascun server può servire un solo task, l'implementazione del server CBS è la seguente:

```
struct cbs_struct {
  unsigned long long int deadline;
  unsigned long int period;
  unsigned long int max_budget;
  unsigned long int period_clock;
  unsigned long int max_budget_clock;
  long long int c;
  int new_task;
  struct task_struct *task;
  struct list_head rlist;
};
```

Il significato dei campi della struttura è abbastanza intuitivo. La variabile `c` è la capacità del server. Il valore delle variabili `period_clock` e `max_budget_clock` è uguale al valore delle variabili `period` e `max_budget` espresso secondo l'unità di misura del TSC:

```
period_clock = us2clock(period);
max_budget_clock = us2clock(max_budget);
```

Nel caso più generale, in cui il server CBS è in grado di gestire più tasks (caso 'multitask'), l'implementazione diventa

```
struct cbs_struct {
  unsigned long long int deadline;
  unsigned long int period;
  unsigned long int max_budget;
  unsigned long int period_clock;
  unsigned long int max_budget_clock;
  long long int c;
  int new_task;
```

```
#ifdef __MULTITASK__
  struct list_head tasks;
  int activations;
#else
  struct task_struct *task;
#endif
  struct list_head rlist;
};
```

La variabile `activations` tiene conto del numero di task attivi (cioè non bloccati) che il server deve servire. Per poter lavorare con l'implementazione 'multitask' è necessario dichiarare la seguente struttura:

```
struct task_list {
  struct task_struct *task;
  struct list_head others;
};
```

Definiamo anche le seguenti variabili globali:

```
static unsigned long long int last_update_time = 0;
static struct cbs_struct *exec = NULL;
static struct list_head ready_list;
```

La variabile `last_update_time` viene uguagliata al tempo corrente prima che un task vada in esecuzione, in modo da poter calcolare la capacità consumata.
Il puntatore `exec` punta in ogni istante il server CBS in esecuzione.
La lista `ready_list` serve per memorizzare i server pronti ad eseguire.

Abbiamo bisogno di un timer che scatti quando il server in esecuzione esaurisce la capacità. Definiamo quindi la seguente variabile globale statica:

```
static struct t_data_struct enforce;
```

Questo timer viene inizializzato mediante

```
enforce.callback = cbs_postpone;
enforce.param = NULL;
sched_init_timer(&enforce);
```

e viene settato con

```
sched_set_timer(S->c, &enforce);
```

Allo scattare di questo timer va in esecuzione la funzione `cbs_postpone()` che provvede a ricaricare la capacità e a spostare la deadline.

6.2 Implementare GRUB a partire da CBS

Supponiamo di avere a disposizione un'implementazione dell'algoritmo CBS, e di voler implementare l'algoritmo GRUB. Questa ipotesi è abbastanza ragionevole, dato che GRUB è stato realizzato due anni dopo CBS, e che l'algoritmo CBS è di più facile implementazione.
Nel nostro caso, poi, disponiamo già dell'implementazione quasi definitiva dell'algoritmo CBS, curata da Luca Abeni nell'ambito del progetto OCERA ([Abe02]).

Cominciamo con il vedere quali modifiche *non* sono necessarie: le somiglianze tra i due algoritmi sono tali da permetterci di riutilizzare buona parte del codice. Il server CBS utilizza i parametri Q_i e T_i, mentre il server GRUB ha bisogno dei parametri U_i e P_i; vediamo qual'è il legame tra queste grandezze.

Poichè sia in CBS che in GRUB utilizziamo l'algoritmo Earliest Deadline First, che impone la priorità dei task in base alla loro deadline, è necessario che la deadline d_i sia la stessa in entrambe le implementazioni.

Per semplificare l'implementazione di GRUB, *vogliamo che l'istante di tempo in cui si verifica l'evento* $V_i = d_i$ *in GRUB coincida con l'istante di tempo in cui si verifica l'evento* $C_i = 0$ *in CBS.* Poichè quando si verificano questi due eventi, la deadline viene incrementata, rispettivamente, di T_i in CBS e di P_i in GRUB, ne deriva che deve essere

$$T_i = P_i$$

É necessario, inoltre, che i server abbiano la stessa banda in entrambe le implementazioni. Dato che la banda del server CBS è $\frac{Q_i}{T_i}$, e la banda del server GRUB è U_i, deve essere

$$U_i = \frac{Q_i}{T_i}$$

Le relazioni trovate permettono di riarrangiare il codice di CBS in modo da ottenere l'implementazione dell'algoritmo GRUB. Notiamo che non occorre tenere traccia del parametro U_i, dato che è facilmente ricavabile mediante i parametri di CBS Q_i e T_i.

Vediamo ora, invece, quali modifiche al codice sono necessarie per poter implementare l'algoritmo GRUB.

1. Quando si verifica l'evento $V_i = d_i$, la deadline d_i viene postposta di P_i, e la capacità C_i viene riportata al valore $Q_i = P_i U_i$.
   Dato che in un intervallo di esecuzione di durata Δt, il virtual time viene incrementato secondo $V_i \mathrel{+}= \frac{U}{U_i}\Delta t$, l'evento $V_i = d_i$ si verificherà nuovamente dopo un intervallo di tempo $\frac{P_i U_i}{U} = \frac{C_i}{U}$.
   Quindi è necessario che il timer `enforce` (che scatta quando bisogna postporre la deadline in seguito ad un esaurimento di capacità) sia settato con il valore $\frac{C_i}{U}$ anzichè con il valore C_i (vedi Figura 6.1):

   ```
   sched_set_timer(llimd(new_exec->c,fix_point,U), &enforce);
   ```

   Per il significato del parametro `fix_point` si veda il successivo punto.

2. É necessario aggiungere una variabile globale statica `U`, inizializzata a 0, che viene incrementata ogni volta che un server passa dallo stato `Inactive` allo stato `ActiveContending`, e decrementata ogni volta che il server entra nello stato `Inactive`:

   ```
   static long int U = 0; /* Global bandwidth*/
   ```

   Poichè vogliamo utilizzare una rappresentazione in virgola fissa, definiamo la seguente costante

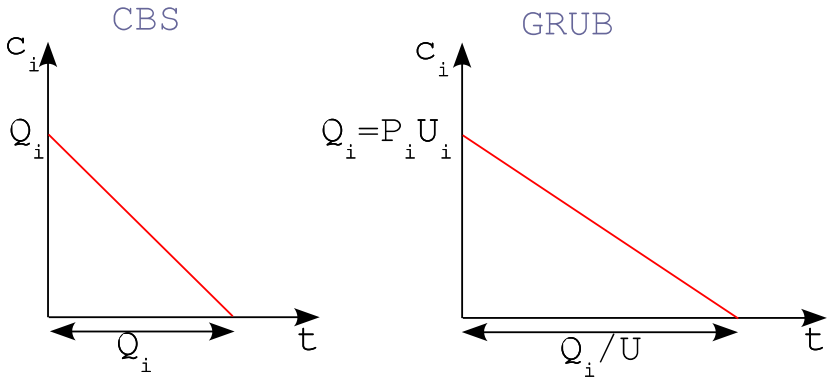

Figura 6.1: Consumo della capacità in CBS e in GRUB

```
#define fix_point 1000 /* Used for fixed point variables */
```

ed utilizziamo un valore di `U` che sia U`*fix_point`. Quando useremo la variabile `U` dovremo ricordarci di ricavare il valore effettivo di essa, mediante la funzione `llimd()`.

Ogni volta che si modifica il valore di U, è necessario fermare il task attualmente in esecuzione, aggiornare il valore della variabile C_i, e settare il timer `enforce` al nuovo valore (Figura 6.2). Per fare questa operazione possiamo utilizzare il seguente codice:

```
long int U_old = U;
unsigned long int consumed_time;
U = ...
sched_stop_timer(&enforce);
if (exec != current->private_data)
   sched_error("GRUB:(cbs_activate) exec!=private_data\n");
if (t >= last_update_time) {
   consumed_time = (unsigned long int)
     llimd ((t - last_update_time), U_old, fix_point);
   if ((exec->c) < consumed_time) {
      exec->c = exec->max_budget_clock;
      exec->deadline += exec->period_clock;
```

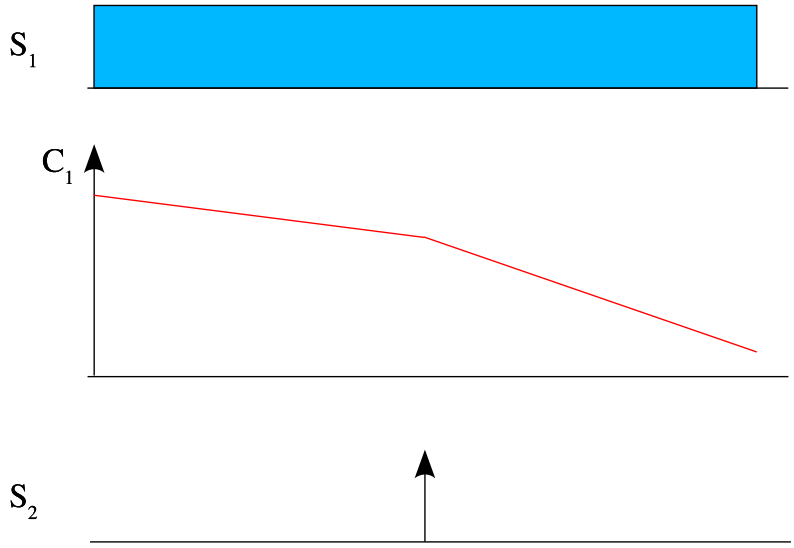

Figura 6.2: Aggiornamento di enforce in seguito al cambio di U

```
    }
    exec->c -= consumed_time;
}
last_update_time = r;
#ifdef __CBS_DEBUG__
    sched_print("last_update_time = %llu\n",last_update_time);
#endif
sched_set_timer(llimd(exec->c, fix_point, U), &enforce);
```

3. Vediamo quale relazione lega il virtual time di GRUB alla capacità C_i di CBS.
   Consideriamo ancora l'istante in cui $V_i = d_i$ (che, per quanto detto finora, coincide con l'istante in cui si esaurisce la capacità in CBS). A seguito di questo evento, la capacità passa dal valore 0 al valore Q_i, mentre la deadline viene spostata dal vecchio valore d_{old} al nuovo valore $d_{new} = d_{old} + P_i$. Supponiamo che il server in questione rimanga attivo. Dopo un breve intervallo di tempo ΔT, la deadline rimane invariata,

mentre il virtual time diventa $V_i += \frac{U}{U_i}\Delta T$ (Figura 6.3).

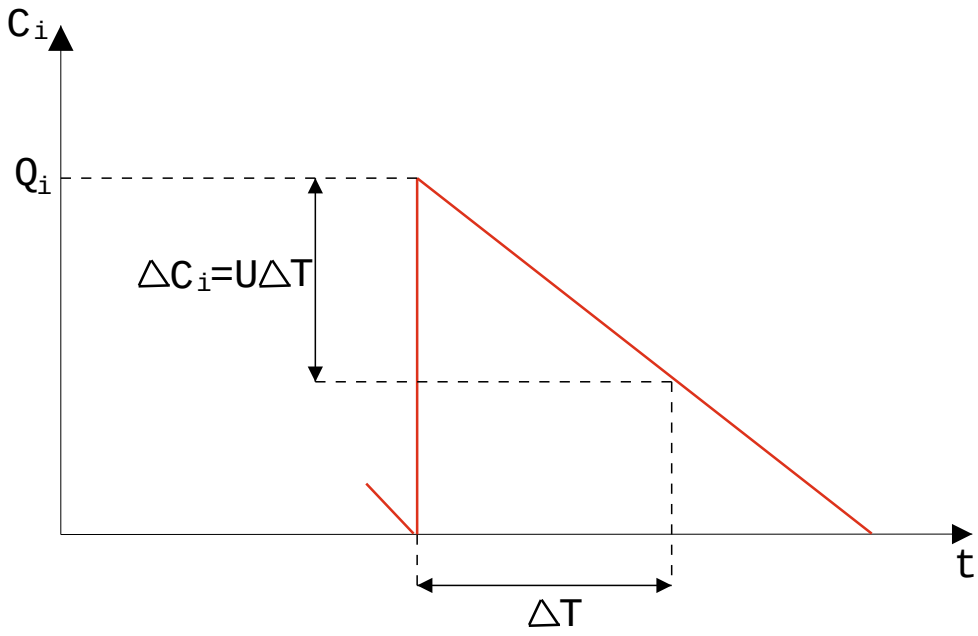

Figura 6.3: Calcolo della relazione tra V_i e C_i

Svolgendo i calcoli, si ottiene

$$V_i = d_{old} + \frac{U}{U_i}\Delta T = d_{new} - P_i + \frac{U}{U_i}\Delta T = d_{new} - P_i + \frac{\Delta C_i}{U_i}$$

$$= d_{new} - P_i + \frac{Q_i}{U_i} - \frac{C_i}{U_i} = d_{new} - \frac{C_i}{U_i} = d_{new} - \frac{C_i}{Q_i}T_i$$

Perciò la relazione che lega il virtual time di GRUB alla capacità C_i di CBS è

$$V_i = d_i - \frac{C_i}{Q_i}T_i$$

Da questo risultato si evince anche che quando si postpone la deadline ($d_i += P_i$) è necessario ricaricare la capacità C_i al valore Q_i, affinché il virtual time non subisca una discontinuità.
Si noti che questa relazione è valida anche nell'istante di esaurimento della capacità (infatti, in tale istante, $V_i = d_i$ per costruzione).

La relazione trovata resta valida anche quando il server non è `Active Contending` (infatti non viene modificata alcuna variabile del server). Questa relazione permette di non dover memorizzare la variabile V_i, dato che può essere ricavata dalle altre variabili del server.

4. Quando un server entra nello stato `ActiveNonContending` aspetta un determinato periodo di tempo dopo il quale, se non è ancora entrato nello stato `ActiveContending`, entra nello stato `Inactive`. Questa transizione di stato si verifica quando $t = V_i$.
   Poichè $V_i = d_i - \frac{C_i}{Q_i}T_i$, detto t' l'istante in cui il server entra nello stato `ActiveNonContending`, l'evento $t = V_i$ si verifica dopo un intervallo di tempo $\Delta t = d_i - t' - \frac{C_i}{Q_i}T_i$.
   Bisogna inserire all'interno della struttura `cbs_struct` un timer (chiamato `inactive`) che scatti nel momento in cui il server passa dallo stato `ActiveNonContending` allo stato `Inactive`. Il timer viene inizializzato con

```
(p->inactive).callback = grub_inactive;
(p->inactive).param = p;
sched_init_timer(&(p->inactive));
p->state = 0; /* not significant value (for debugging) */
```

   e viene settato con

```
long long int inactive_time =
     t->deadline - time - llimd(S->c, S->period, S->max_budget);
sched_set_timer(inactive_time, &(S->inactive));
```

   Allo scattare del timer, va in esecuzione la funzione `grub_inactive()`, che provvede ad effettuare la transizione di stato del server da `ActiveNon Contending` a `Inactive`, e ad aggiornare il valore della variabile `U`. Il codice equivalente alle linee 28-31 dello pseudocodice discusso nel terzo capitolo è il seguente:

```
/* This function starts when a Server becomes inactive */
```

```
static void grub_inactive(unsigned long long int time, void* v)
{
   long int U_old = U;
   struct cbs_struct* t;
   if (v == NULL)
     sched_error("ERROR: parameter of grub_inactive is NULL\n");
   t = (struct cbs_struct*) v;
   if (t->state == 2) {
      /* U-=Ui */
      U-=llimd(t->max_budget,fix_point,t->period);
#ifdef __GRUB_DEBUG__
      sched_print("GRUB :Inactivating task\n");
      sched_print("GRUB :U=%ld->%ld\n", U_old, U);
#endif
      if (U < 0) {
        sched_error("GRUB ERROR(grub_inactive): U =%ld< 0\n",U);
        U = 0;
      }
      t->state= 3; /* Server becomes inactive */
      /* Change the execution time for exec */
      if (exec != NULL && exec!= t) {
        unsigned long int consumed_time;
        sched_stop_timer(&enforce);
#ifdef __GRUB_DEBUG__
        if (exec != current->private_data)
          sched_print("GRUB: exec!=private_data\n");
#endif
        if (time >= last_update_time) {
           consumed_time = (unsigned long int)
             llimd ((time - last_update_time), U_old, fix_point);
           if ((exec->c) < consumed_time) {
             exec->c = exec->max_budget_clock;
             exec->deadline += exec->period_clock;
           }
           exec->c -= consumed_time;
```

```
        }
        last_update_time = time;
#ifdef __GRUB_DEBUG__
        sched_print("reSetting timer enforce with %lu\n",
          clock2us((unsigned long long int)
           ((exec->c)*fix_point/U)));
#endif
        sched_set_timer(llimd(exec->c, fix_point, U), &enforce);
      }
   }
}
```

Se, mentre un server è nello stato `ActiveNonContending` arriva una nuova richiesta da parte di un task appartenente a quel server, bisogna fermare il timer `inactive` e portare il server nello stato `ActiveContending`. Il codice equivalente alle linee 13-17 dello pseudocodice discusso nel terzo capitolo è il seguente:

```
if (t->state == 2) {
   /* Server was ActiveNonContending */
   /* Since Vi = deadline-(c/bandwidth) we just postpone
   /* deadline of (period-(c/bandwidth)) */
   t->deadline += (unsigned long long int)
     ((t->period_clock) - llimd(t->c,t->period,t->max_budget));
   t->c = t->max_budget_clock;
   sched_stop_timer(&(t->inactive));
#ifdef __GRUB_DEBUG__
   sched_print("GRUB:Stopping timer inactive\n");
   sched_print("GRUB:Activating an ActiveNonContending Server\n");
#endif
}
t->state = 1; /* Server becomes ActiveContending */
```

5. Bisogna aggiungere nella struttura dati del server `cbs_struct` un campo che indichi lo stato in cui si trova il server:

`int state;`

Questa variabile intera indica se il Server è nello stato `ActiveContending` (`state=1`), nello stato `ActiveNonContending` (`state=2`), oppure nello stato `Inactive`(`state=3`).

In realtà, questa variabile non è strettamente necessaria. Lo stato di un server non attivo può essere infatti ricavato tramite il test di CBS : se $C_i > (d_i - t) * U_i$ allora il server è nello stato `Inactive`, altrimenti è nello stato `ActiveNonContending`.
Tuttavia, bisogna considerare che il timer `inactive` non scatta sempre nel preciso istante in cui dovrebbe scattare (ricordiamo che la coda dei timer viene controllata dal sistema operativo ogni 10ms). Perciò esistono casi in cui il server dovrebbe essere nello stato `Inactive`, ma è ancora nello stato `ActiveNonContending` perché il timer `inactive` non è ancora scattato. Per evitare di aggiornare erroneamente la variabile U, conviene tenere traccia dello stato del server.

L'implementazione del server per GRUB è, quindi, la seguente:

```
struct cbs_struct {
  unsigned long long int deadline;
  unsigned long int period;
  unsigned long int max_budget;
  unsigned long int period_clock;
  unsigned long int max_budget_clock;
  long long int c;
  int new_task;
#ifdef __MULTITASK__
  struct list_head tasks;
  int activations;
#else
  struct task_struct *task;
#endif
  struct list_head rlist;
```

```
#ifdef __GRUB__
  int state;
  struct t_data_struct inactive;
#endif
};

#ifdef __GRUB__
#define fix_point 1000 /* Used for fixed point variables (U) */
static long int U = 0; /* Global bandwidth*/
#endif
```

6.3 Implementare l'Hard Reservation

Nel quarto capitolo è stato descritto dettagliatamente il problema che è intrinsecamente presente nell'algoritmo GRUB. É stata peraltro dimostrata la correttezza della regola di Hard Reservation, che permette di superare il problema semplicemente 'addormentando' i task senza rilasciare la banda che essi detengono.

Consideriamo il server in esecuzione S_i. Dobbiamo fare in modo che quando la deadline d_i del server viene postposta in seguito al verificarsi dell'evento $V_i = d_i$, il server S_i si 'addormenti' fino al momento in cui si verifica l'evento $t = V_i$.

Poichè l'evento $V_i = d_i$ si verifica quando il server esaurisce la propria capacità C_i, se t_0 è l'istante in cui S_i esaurisce la capacità C_i, a tale istante S_i deve entrare nello stato `Suspended`.
S_i deve restare in questo stato fino a quando si verifica l'evento $t = V_i$.
Ricordiamo, che vale la relazione

$$V_i = d_i - \frac{C_i}{Q_i} T_i$$

Perciò dire $t = V_i$ e $C_i = 0$, equivale a dire $t = d_i$. Quindi, indicato con d_{old} il valore della deadline d_i un istante prima di t_0, il server S_i deve restare nello stato `Suspended` per un tempo pari a $(d_{old} - t_0)$.

Per realizzare questa regola, possiamo avvalerci del timer `t_data_struct` che ci siamo già preoccupati di implementare nel precedente capitolo. Al momento in cui il server entra nello stato `Suspended`, viene settato un timer in modo che scatti dopo un tempo pari a $(d_{old} - t_0)$, e viene rimosso il server dalla lista dei server pronti. Allo scattare del timer va in esecuzione una funzione che provvede a spostare il server nella lista dei server pronti, e a forzare un cambio di contesto.

Notiamo che più server possono essere contemporaneamente nello stato `Suspended`, perciò è necessario prevedere un timer per ciascuno di essi. Si capisce perciò che il timer per la sospensione (che chiamiamo `reactive`) deve essere inserito direttamente all'interno della struttura `cbs_struct`:

```
struct cbs_struct {
  unsigned long long int deadline;
  unsigned long int period;
  unsigned long int max_budget;
  unsigned long int period_clock;
  unsigned long int max_budget_clock;
  long long int c;
  int new_task;
#ifdef __MULTITASK__
  struct list_head tasks;
  int activations;
#else
  struct task_struct *task;
#endif
  struct list_head rlist;
#ifdef __GRUB__
  int state;
  struct t_data_struct inactive;
#endif
#ifdef __SLEEP__
  /* timer to become active after being suspended */
  struct t_data_struct reactive;
```

```
#endif
};
```

All'interno della funzione `cbs_sleep_postpone()`, la quale ha il compito di postporre la deadline in seguito ad un esaurimento di capacità, inseriamo il seguente codice (equivalente alle linee 32-36 dello pseudocodice del quarto capitolo):

```
/* This function starts when the timer enforce fires */
static void cbs_sleep_postpone(unsigned long long int time,void *dummy)
{
  unsigned long long int old_deadline;
//....
  old_deadline = t->deadline;
#ifdef __CBS_DEBUG__
  sched_print("Setting timer reactive %d with %lu\n", t->reactive.id,
              clock2us(old_deadline - time));
#endif
  list_del(&(t->rlist)); /* Remove the server from the list */
  sched_set_timer(old_deadline - time, &(t->reactive));
  cbs_schedule(time);
//..
}
```

Allo scattare del timer, va in esecuzione la seguente funzione (equivalente alle linee 37-40 dello pseudocodice del quarto capitolo):

```
/* This function starts when a Server rebecomes active after being
 * blocked (when the timer reactive fires)
 */
static void cbs_sleep_reactivate(unsigned long long int time,void* v)
{
  struct cbs_struct* t;
  if (v== NULL)
     sched_error("CBS ERROR: parameter of cbs_sleep_reactivate
                  is NULL\n");
#ifdef __CBS_DEBUG__
```

```
  sched_print("Starting cbs_sleep_reactivate\n");
#endif
  t = (struct cbs_struct*) v;
  if (edf_list_add(t, 0)) {
     /* We need a context switch */
     if (exec) {
        /* Stop the executing server */
        if (update_used_time(exec, time)) {
           update_priorities(exec);
        }
     }
     last_update_time = time;
     cbs_schedule(time);
  }
}
```

Il timer `reactive` deve essere inizializzato nel modo seguente:

```
#ifdef __SLEEP__
  (p->reactive).callback = cbs_sleep_reactivate;
  (p->reactive).param=p;
  sched_init_timer(&(p->reactive));
#endif
```

É necessario ricordarsi di inizializzare il timer `enforce` in modo che al suo scattare vada in esecuzione la funzione `cbs_sleep_postpone()` e non la classica `cbs_postpone()`:

```
#ifdef __SLEEP__
  enforce.callback = cbs_sleep_postpone;
#else
  enforce.callback = cbs_postpone;
#endif
```

Capitolo 7

Il risparmio energetico

7.1 Obiettivo

Nel terzo capitolo è stato presentato l'algoritmo GRUB, nel quale la banda U_i rappresenta la capacità totale del processore che deve essere dedicata ai task del server S_i. Al server S_i, sembra, cioè, che i propri job siano in esecuzione su un processore virtuale dedicato con velocità pari a U_i volte la velocità del processore attuale.

Vogliamo modificare l'implementazione dell'algoritmo GRUB vista nel precedente capitolo in modo che in ogni istante la frequenza di clock della CPU sia proporzionale al valore della variabile `U`. In questo modo, quando il carico del sistema è sufficientemente basso, il consumo dell'hardware è ridotto.

7.2 Hardware utilizzato

Per implementare il risparmio energetico abbiamo usato una macchina *Intrinsyc CerfCube 250* (Figura 7.1) con le seguenti caratteristiche:

- Memoria:

  32 MB Flash

  64 MB SDRAM (100 MHz)

Figura 7.1: Intrinsyc CerfCube 250

2 EEPROM

- Data Connectivity:

  10/100 Ethernet

  USB 1.1 Tipo B

- Alimentazione: 5V DC regulated, 400mA (picco di 800-900mA)

- Software:

  Linux Kernel 2.4.18

  I-Linux 4.3

Il microprocessore di questa macchina, è un Intel PXA250 a 400 MHz (vedi Figura 7.2). Questo Application Processor ha elevate prestazioni, e racchiude la *microarchitettura Intel Xscale* che permette una modifica on-the-fly della frequenza di clock, ed una sofisticata gestione del consumo in modo da fornire un rapporto MIPs/mW leader nel campo industriale. Questo processore ha, inoltre, le seguenti caratteristiche:

- Dimensioni: 17x17mm

- 32 bit

- Memory bus a 32 bit

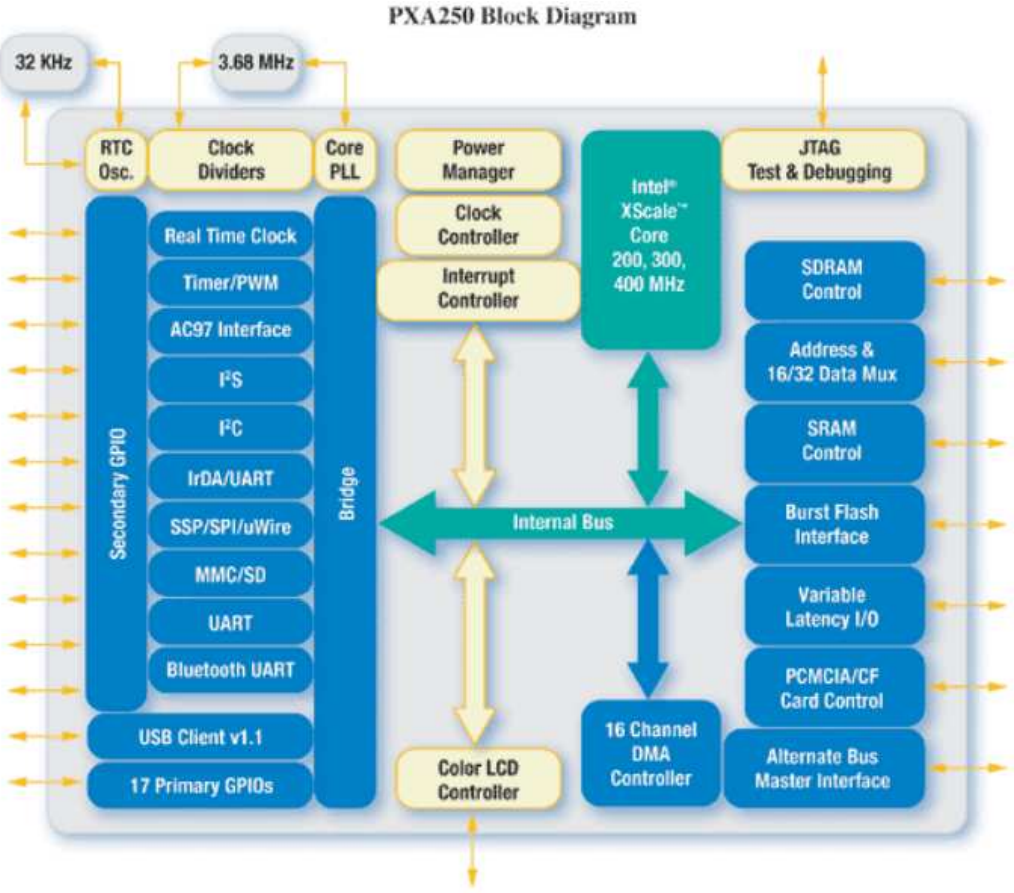

Figura 7.2: Processore Intel PXA250

- Compatibile con l'architettura ARM versione 5TE
- Codice compatibile con il processore Intel SA-1110
- Tecnologia RISC superpipelined a 0.18 micron
- 32 KB di Data Cache (2 KB di Mini Data Cache) e 32 KB di Instruction Cache (2KB di Mini Instruction Cache)

7.2.1 Frequenza di clock

Il processore PXA250 offre la possibilità di cambiare dinamicamente la frequenza di clock della CPU. In particolare, il processore può essere in uno dei seguenti stati:

1. Turbo Mode: il Core funziona alla frequenza di picco. In questa modalità conviene fare pochi accessi alla memoria esterna perché il Core, essendo più veloce, è costretto ad aspettare.

2. Run Mode: il Core funziona alla sua frequenza normale. In questa modalità si assume che il Core faccia frequenti accessi alla memoria esterna, perciò è conveniente che funzioni ad una frequenza inferiore a quella del Turbo Mode, per ottimizzare il trade-off consumo/performance.

3. Idle Mode: il Core non viene alimentato (quindi nemmeno i clock interni), ma il resto del sistema rimane pienamente operativo. Questa modalità è usata durante brevi momenti di inattività, nei quali il sistema esterno deve continuare le operazioni. Si esce da questa modalità attraverso un interrupt.

4. Sleep Mode: pone il processore nel suo stato di minore consumo (e non ne mantiene lo stato interno), ma mantiene lo stato di I/O, il Real Time Clock (o RTC), i vari clock, ed il Power Manager. Svegliarsi dallo Sleep Mode richiede il reboot del sistema, dato che molti stati interni vengono persi.

Il registro nel quale viene scritta la frequenza di clock desiderata prende il nome di CCCR (*Core Clock Configuration Register*), si trova all'indirizzo `0x41300000`, ed è anch'esso definito all'interno del file `<include/asm-arm/arch-pxa/pxa-regs.h>`.
Il CCCR controlla la frequenza di clock del Core, dalla quale vengono derivate anche le frequenze di clock dei controllori della memoria, del LCD e del DMA. In questo registro sono specificati i seguenti parametri:

- Moltiplicatore dalla frequenza del cristallo alla frequenza della memoria (L)

- Moltiplicatore dalla frequenza della memoria alla frequenza della CPU in Run-Mode (M)

- Moltiplicatore dalla frequenza della CPU in Run-Mode a quella in Turbo-Mode (N)

Il valore di L è scelto basandosi sui vincoli della memoria esterna o del LCD, e può essere costante, mentre i valori di M e N cambiano per permettere un cambiamento delle frequenze della CPU in Run-Mode ed in Turbo-Mode senza dover cambiare i settaggi della memoria.
Il valore di M è scelto basandosi sui vincoli di banda del bus e sui requisiti minimi di performance del Core.
Il valore di N è scelto in base ai vincoli di performance del Core durante i picchi.

Il registro CCCR è un registro a 32 bit, suddiviso nel modo seguente ([Intel02]):

Bit	Utilizzo
31:10	Riservati (la lettura fornisce un valore indefinito, mentre la scrittura deve essere sempre fatta con zero)
9:7	Moltiplicatore N <frequenza in Turbo mode> = <frequenza in Run mode> * N 000 Riservato 001 Riservato 010 Moltiplicatore = 1 011 Moltiplicatore = 1.5 100 Moltiplicatore = 2 101 Riservato 110 Moltiplicatore = 3 111 Riservato

6:5	Moltiplicatore M <frequenza in Run mode> = <frequenza della memoria> * M 00 Riservato 01 Moltiplicatore = 1 10 Moltiplicatore = 2 11 Riservato
4:0	Moltiplicatore L <frequenza della memoria> = <frequenza del cristallo> * L La frequenza del cristallo è sempre 3.6864 MHz 00000 Riservato 00001 Moltiplicatore = 27 (frequenza della memoria = 99.53 MHz) 00010 Moltiplicatore = 32 (frequenza della memoria = 117.96 MHz) 00011 Moltiplicatore = 36 (frequenza della memoria = 132.71 MHz) 00100 Moltiplicatore = 40 (frequenza della memoria = 147.46 MHz) 00101 Moltiplicatore = 45 (frequenza della memoria = 165.89 MHz) 00110 Riservato :::: 11111 Riservato

Bisogna prestare molta attenzione quando si impostano i valori dei diversi clock, perché non viene fatto alcun controllo sulla frequenza di clock del processore, ed è molto facile impostare per errore una frequenza troppo elevata (i.e. superiore a 400 MHz).

Per modificare la frequenza di clock del sistema, si utilizza il registro CCLKCFG, che è il registro numero 6 del coprocessore 14 (addetto alla gestione del clock e del risparmio energetico).
CCLKCFG è un registro a 32 bit, usato per entrare nelle modalità Turbo Mode e Frequency Change Sequence, ed è suddiviso nel modo seguente:

Bit	Utilizzo
31:2	Riservati (la lettura fornisce un valore indefinito, mentre la scrittura deve essere sempre fatta con zero)
1	bit FCS (Frequency Change Sequence) 0 Non entrare nella modalità Frequency Change Sequence 1 Entra nella modalità Frequency Change Sequence
0	bit Turbo 0 Non entrare nella modalità Turbo Mode / Esci dalla modalità Turbo Mode 1 Entra nella modalità Turbo Mode

Il registro CCLKCFG non può essere acceduto direttamente. Per leggere, o settare, questi due bit è necessario usare alcune istruzioni Assembler. Per leggere il valore del registro (e metterlo nel registro `R0`) si usa

```
MCR p14, 0, R0, c6, c0, 0
```

Per copiare il contenuto del registro `R0` nel registro CCLKCFG, si usa

```
MRC p14, 0, R0, c6, c0, 0
```

Per assicurarsi che il bit Turbo non cambi quando si entra nella modalità Frequency Change Sequence, il software deve fare una read-modify-write.

La modalità Frequency Change Sequence viene usata per cambiare la frequenza di clock del processore. Durante questa sequenza, i clock della CPU, del controller della memoria, del controller LCD e del DMA, si fermano. Le altre unità periferiche continuano a funzionare.

Questa modalità viene generalmente utilizzata per impostare le frequenze di clock del processore e della memoria ad un valore diverso da quello di default del boot iniziale. Tuttavia, può anche essere usata come meccanismo per ridurre il consumo, permettendo al processore di funzionare alla frequenza minima necessaria.

Prima di cominciare la sequenza di cambio di frequenza, il software deve completare i seguenti passi:

1. Configurare il controller della memoria per assicurarsi che il contenuto della memoria SDRAM venga mantenuto durante il cambio di frequenza.

2. Disabilitare il controller LCD, o configurarlo in modo da evitare gli effetti di una interruzione nel clock e nei dati provenienti dal processore

3. Configurare le unità periferiche per gestire una mancanza del servizio DMA fino ad un massimo di 500 microsecondi. Se una periferica non può funzionare per 500 microsecondi senza DMA, allora va disabilitata.

4. Programmare il registro CCCR secondo le frequenza desiderate

A questo punto, per invocare la sequenza di cambio di frequenza, il software deve settare i bit FCS e TURBO nel registro CCLKCFG. Non appena vengono settati questi due bit,

1. il clock della CPU si ferma e gli interrupt verso la CPU vengono catturati

2. il controller della memoria completa tutte le transazioni outstanding presenti nel buffer e provenienti dalla CPU. Le nuove transazioni dai controller LCD o DMA vengono ignorate.

3. il controller della memoria pone la SDRAM in modalità di auto-refresh.

Al termine della sequenza di cambio di frequenza

1. Il clock della CPU riparte. Gli interrupt verso la CPU non vengono più catturati.

2. Il bit FCS non viene azzerato automaticamente. Per prevenire un ritorno accidentale nella modalità Frequency Change Sequence, il software non deve azzerare immediatamente questo bit. Il bit deve essere azzerato durante la successiva scrittura nel registro.

3. Possono venir scritti valori in CCCR, ma essi vengono ignorati finché non si rientra nella sequenza di cambio di frequenza

4. La SDRAM esce dalla modalità di auto-refresh ed entra nella modalità idle.

Bisogna fare molta attenzione a non superare la frequenza di clock massima del sistema (i.e. 400 MHz), perché non viene fatto alcun controllo da parte dell'hardware della macchina.

7.3 Implementazione

7.3.1 Trattare interi a 64 bit sul processore PXA250

Essendo il processore PXA250 a 32 bit, sorge nuovamente il problema di trattare le variabili temporali a 64 bit su una architettura che non prevede questa funzionalità. Ancora una volta, si utilizza la funzione `llimd()` fornita dal gruppo di ricerca di RTAI.

Dato che non esiste una definizione ad hoc per il processore PXA250 (infatti questo processore è relativamente nuovo), possiamo usare la definizione per il processore SA-1110:

```
/* These functions come from RTAI (http://www.rtai.org) */

/* not all ARM cores support 32bit x 32bit = 64bit native ... */
static inline unsigned long long ullmul(unsigned long m0,
                                        unsigned long m1)
{
        unsigned long long res;
        res = (unsigned long long)m0 * (unsigned long long)m1;
        return res;
}

static inline unsigned long long ulldiv(unsigned long long ull,
                                        unsigned long uld,
unsigned long *r)
{
unsigned long long q = ull/(unsigned long long) uld;
*r = (unsigned long) (ull - q * (unsigned long long) uld);
return q;
}

static inline unsigned long long llimd(unsigned long long ull,
                                       unsigned long mult,
       unsigned long div)
{
```

```
unsigned long long low, high, q;
unsigned long r;

low  = ullmul(((unsigned long *)&ull)[0], mult);
high = ullmul(((unsigned long *)&ull)[1], mult);
q = ulldiv(high,div,&r) << 32;
high = ((unsigned long long) r) << 32;
q += ulldiv( high + low, div , &r);
return (r + r) < div ? q : q + 1;
}
```

Bisogna notare che, a differenza dell'implementazione vista per l'architettura `i386` (dove la divisione era stata realizzata con una moltiplicazione ed uno shift), in questo caso la funzione non è ottimizzata.

7.3.2 Il tempo nel processore PXA250

I blocchi funzionali del processore PXA250 sono guidati da clock derivati da un cristallo a 3.6864 MHz. Uno di questi blocchi funzionali è l'*OS Timer*, che fornisce un contatore di riferimento alla suddetta frequenza.

L'architettura del processore PXA250 non prevede il registro Time Stamp Counter a 64 bit, ma prevede un altro tipo di counter, che prende il nome di OSCR (*OS timer Counter Register*). A differenza del registro TSC dell'architettura i386, questo è un registro a 32 bit, il cui indirizzo (`0x40A00010`) è definito nel file `<include/asm-arm/ arch-pxa/pxa-regs.h>`, nel modo seguente:

```
#define OSCR  __REG(0x40A00010)  /* OS Timer Counter Register */
```

La funzione REG è una macro definita nel file `<include/asm-arm/arch-pxa/hardware.h>`:

```
/*
 * This __REG() version gives the same results as the one above,
 * except that we are fooling gcc somehow so it generates far
 * better and smaller assembly code for access to contigous
 * registers.  It's a shame that gcc  doesn't guess this by itself.
```

```
 */
#include <asm/types.h>
typedef struct { volatile u32 offset[4096]; } __regbase;
# define __REGP(x)  ((__regbase *)((x)&~4095))->offset[((x)&4095)>>2]
# define __REG(x)   __REGP(io_p2v(x))
```

La funzione per leggere il valore del registro OSCR è semplicemente

```
static inline unsigned long long sched_read_clock(void)
{
  return (unsigned long long) OSCR;
}
```

All'interno del file `<include/linux/timex.h>` sono definite le seguenti variabili:

```
/* LATCH is used in the interval timer and ftape setup. */
#define LATCH  ((CLOCK_TICK_RATE + HZ/2) / HZ)  /* For divider */

extern long tick; /* timer interrupt period */
```

Queste variabili permettono di passare dal valore espresso secondo OSCR al valore in microsecondi (e, quindi, anche in jiffies). Le conversioni di tempo possono essere fatte attraverso le seguenti funzioni:

```
static inline unsigned long long int us2clock(unsigned long u)
{
  return llimd(u, LATCH, tick);
}

static inline unsigned long int clock2ms(unsigned long long c)
{
  return (unsigned long int)(llimd(c, tick, LATCH)/1000);
}

static inline unsigned long int clock2us(unsigned long long c)
{
  return (unsigned long int) llimd(c, tick, LATCH);
```

```
}

static inline unsigned long int clock2jiffies(unsigned long long c)
{
  unsigned long long int tmp;
  tmp = llimd(c, tick, LATCH);
  return (unsigned long int) llimd(tmp, HZ, 1000000);
}
```

7.3.3 Risparmio energetico

Dovendo scegliere come implementare il risparmio energetico, e quali frequenze utilizzare, abbiamo deciso di mantenere la frequenza di clock della memoria costante, e di variare la frequenza di clock del processore su tre livelli (100 MHz, 200 MHz, 400 MHz). Abbiamo ritenuto inutile creare numerosi livelli di frequenze alle quali far andare il processore. Se il carico del sistema U fosse costante, infatti, con un maggior numero di livelli saremmo in grado di trovare una frequenza di clock del sistema più appropriata per soddisfare il trade-off consumo/prestazioni. Tuttavia, il nostro sistema è rivolto ad un sistema operativo, perciò ad un sistema con un carico che varia molto dinamicamente, anche a causa dei processi stessi di Linux, che eseguono per breve tempo e poi si bloccano. In questo caso il piccolo beneficio che potremmo trarre da un numero elevato di livelli, non sarebbe tale da giustificare l'incredibile overhead che comparirebbe nel sistema.

Utilizzando i tre livelli di frequenza descritti (100, 200 e 400 MHz) siamo in grado di raggiungere sia la frequenza minima del processore (minimo consumo) sia quella massima (massime prestazioni), spaziando all'interno del range delle frequenze che esso mette a disposizione. In particolare, la frequenza della memoria viene mantenuta a 99.53 MHZ (L=00001), mentre il processore può trovarsi in uno dei seguenti stati:

Consumo	Prestazioni	M	Run Mode	N	Turbo Mode	Turbo
Basso	Basse	01	99.53 MHz	-	-	NO
Medio	Medie	10	199.06 MHz	-	-	NO
Alto	Alte	10	199.06 MHz	100	398.12 MHz	SI

La nostra implementazione è tale che quando avviene un aumento del carico del sistema ($U+ = U_i$) la frequenza di clock viene cambiata immediatamente. Infatti, anche se c'è il rischio che si tratti di un breve picco transitorio, non possiamo permetterci di aspettare un pò di tempo per vedere l'evolversi della situazione: le conseguenze di un tale comportamento, nel caso in cui non si trattasse di un picco di breve durata, sarebbero terribili.

Nel caso in cui, invece, si abbia una riduzione del carico ($U- = U_i$), vogliamo evitare di abbassare immediatamente la frequenza di clock. Se così facessimo, infatti, e la riduzione fosse di breve durata, dovremmo cambiare frequenza due volte, introducendo nel sistema un notevole overhead. Definiamo perciò la variabile `PWR_TIMEOUT` che specifica il numero di secondi da attendere prima di scalare effettivamente la frequenza verso un valore più basso. Se, nel frattempo, il carico del sistema aumenta, e la frequenza prevista non è tale da soddisfare la richiesta di banda, allora si annulla il cambio di frequenza previsto.

Abbiamo indicato con la variabile `NUM_PWR_LEVELS` il numero di livelli di frequenze che abbiamo intenzione di usare (nel nostro caso tre). Ciascun livello è descritto da una struttura `pwr_level` con la seguente definizione:

```
struct pwr_level {
   int bandwidth;
   int run_mode; /* 1 or 2 */
   int turbo_mode; /* 1 or 2 */
   int selected;   /* shows if the timer is set */
   struct t_data_struct pwr_timer;
};
```

Il primo parametro (`bandwidth`) indica la banda massima che il livello è in grado di supportare. Anche il valore di questa variabile è normalizzato

con il parametro `fix_point`, come già discusso nel precedente capitolo. Nel nostro caso, se `fix_point = 1000`, il valore di questa variabile sarà 250 per la frequenza di 100 MHz, 500 per la frequenza di 200 MHz e 1000 per la frequenza di 400 MHz (supponendo che le prestazioni crescano linearmente con la frequenza di clock del processore).
Il secondo ed il terzo parametro indicano, rispettivamente, il moltiplicatore per il Run Mode (ossia M) ed il moltiplicatore per il Turbo Mode (ossia N). Quando c'è una diminuzione del carico del sistema, e viene deciso di portare il sistema ad un livello di frequenza inferiore, viene settato il timer `pwr_timer` di quel livello, e viene settata la variabile `selected` per evitare di settare nuovamente il timer nel caso di un'oscillazione del carico. Allo scattare del timer, va in esecuzione la funzione `pwr_callback()` che ha il compito di modificare la frequenza del processore. Se nel frattempo, l'utilizzazione del sistema cresce nuovamente, e si vede che il nuovo livello non sarebbe in grado di soddisfare le nuove esigenze, allora il timer `pwr_timer` viene fermato, e la variabile `selected` resettata.

Per conoscere quale livello è utilizzato ad ogni istante, abbiamo utilizzato un puntatore alla struttura `pwr_level`:

```
struct pwr_level* current_pwr_level;
```

Il codice per il risparmio energetico è il seguente:

```
#define NUM_PWR_LEVELS 3
#define PWR_TIMEOUT 3 /* Number of seconds to wait before scaling
                       * frequency */

struct pwr_level {
   int bandwidth;
   int run_mode; /* 1 or 2 */
   int turbo_mode; /* 1 or 2 */
   int selected;   /* shows if the timer is set */
   struct t_data_struct pwr_timer;
};
```

```
struct pwr_level pwr_levels [NUM_PWR_LEVELS];
struct pwr_level* current_pwr_level;

void pwr_change_frequency(int run_mode_mult, int turbo_mode_mult)
{
   unsigned int r;
   r = CCCR;
   sched_print("GRUB: CCCR=%u (pwr_change_frequency)\n",r);

   /* Set CCCR for run mode */
   if (run_mode_mult == 1) {
      r |= 0x0020;
      r &= 0x11B1;
   }
   else {
      r |= 0x0040;
      r &= 0x11D1;
   }

   /* Set CCCR for turbo mode */
   if (turbo_mode_mult == 1) {
      r |= 0x0100;
      r &= 0x1D71;
   }
   else if (turbo_mode_mult == 2) {
      r |= 0x0200;
      r &= 0x1E71;
   }

   CCCR = r;

   /* Read the value of CCLKCFG */
   __asm__ __volatile__ ("mrc p14,0,%0,c6,c0,0" :"=r" (r):);

   sched_print("GRUB: previous CCLKCFG=%u\n",r);
```

```
   /* Set the turbo mode bit */
   if (turbo_mode_mult == 2)
     r |= 0x0001;
   else
     r &= 0x111E;

   /*Set the frequency change bit */
   r |= 0x0002;

   __asm__ __volatile__ ( "mcr p14,0,%0,c6,c0,0"::"r"(r));

  sched_print ("Frequency changed run:%d turbo:%d\n", run_mode_mult,
               turbo_mode_mult);
}


void pwr_callback(unsigned long long int time, void * param)
{
   struct pwr_level* new_level = (struct pwr_level*) param;
   sched_print("GRUB: (pwr_callback)\n");
   new_level->selected = 0;
   pwr_change_frequency (new_level->run_mode, new_level->turbo_mode);
   current_pwr_level = new_level;
}


void pwr_init ()
{
   int i;
   unsigned int r;
   sched_print("GRUB: (pwr_init)\n");

   /* Set memory frequency in CCCR*/
   r = CCCR;
```

```
   r |= 0x0001;
   r &= 0x11E1;
   CCCR = r;

   for (i=0; i< NUM_PWR_LEVELS; i++) {
      pwr_levels[i].selected = 0;
      sched_init_timer(&(pwr_levels[i].pwr_timer));
      pwr_levels[i].pwr_timer.param = &(pwr_levels[i]);
      pwr_levels[i].pwr_timer.callback = pwr_callback;
   }
   pwr_levels[0].bandwidth = 250;
   pwr_levels[0].run_mode = 1;
   pwr_levels[0].turbo_mode = 1;
   pwr_levels[1].bandwidth = 500;
   pwr_levels[1].run_mode = 2;
   pwr_levels[1].turbo_mode = 1;
   pwr_levels[2].bandwidth = 1000;
   pwr_levels[2].run_mode = 2;
   pwr_levels[2].turbo_mode = 2;

   current_pwr_level = &(pwr_levels[0]);
   sched_print("GRUB: current_level = %d (pwr_init)\n",
               current_pwr_level->bandwidth);
   pwr_change_frequency (current_pwr_level->run_mode,
                         current_pwr_level->turbo_mode);
}

void pwr_set(int level)
{
   pwr_levels[level].selected = 1;
   sched_set_timer( us2clock(PWR_TIMEOUT*1000000),
                   &(pwr_levels[level].pwr_timer));
}

int pwr_up(long int U_new)
```

```
{
   int i;
   sched_print("GRUB: U=%ld (pwr_up)\n",U_new);
   /* See if we have to stop some timer... */
   for (i = 0 ; i < NUM_PWR_LEVELS ; i++)
      if ((pwr_levels[i].selected == 1) &&
          ((pwr_levels[i].bandwidth) < U_new) ) {
         pwr_levels[i].selected = 0;
         sched_stop_timer(&(pwr_levels[i].pwr_timer));
      }
   /* See if we can continue with this frequency */
   if (current_pwr_level->bandwidth >= U_new) {
      sched_print("GRUB: Can use old level (current =%d)\n",
                  current_pwr_level->bandwidth);
      return 0;
   }
   /* We must change frequency */
   for (i=0; i < NUM_PWR_LEVELS; i++)
      if (pwr_levels[i].bandwidth >= U_new) {
            if (pwr_levels[i].selected == 1) {
               pwr_levels[i].selected = 0;
               sched_stop_timer (&(pwr_levels[i].pwr_timer));
            }
            pwr_change_frequency (pwr_levels[i].run_mode,
                                  pwr_levels[i].turbo_mode);
            current_pwr_level = &(pwr_levels[i]);
            return 1;
      }
   return 2;
}


void pwr_down(long int U_new)
{
   int i;
```

```
   sched_print("GRUB: U=%ld (pwr_down)\n",U_new);
   /* Look if we can change frequency */
   for (i=0; i < NUM_PWR_LEVELS; i++)
      if (pwr_levels[i].bandwidth >= U_new) {
         if ( (&(pwr_levels[i]) != current_pwr_level) &&
              (pwr_levels[i].selected == 0) ) {
            pwr_levels[i].selected = 1;
            pwr_set(i);
         }
         break;
   }
}
```

É inoltre necessario modificare il codice di GRUB, facendo in modo che ad ogni aumento e riduzione di banda, vadano in esecuzione, rispettivamente, le funzioni `pwr_up(U)` e `pwr_down(U)`, dove `U` è il nuovo valore dell'utilizzazione totale del sistema.

Capitolo 8

Installazione e Test

8.1 Installazione

Adesso che abbiamo visto come è stato implementato lo scheduler real-time, vediamo come è possibile installarlo su un'architettura che abbia come sistema operativo Linux.

É necessario eseguire nell'ordine le seguenti operazioni:

1. Applicare al kernel di Linux la 'Generic Scheduler Patch'
2. Applicare al kernel di Linux la 'Preemption Patch'. Questa operazione è facoltativa, ma è fortemente raccomandata: ricordiamo, infatti, che l'applicazione di questa patch permette di ottenere tempi di risposta migliori per i task eseguiti in modalità real-time.
3. Compilare ed installare il kernel di Linux
4. Compilare ed installare lo scheduler
5. Compilare il software per il test e per l'esecuzione dei task real-time

Vediamo alcune di queste operazioni in modo più dettagliato.

8.1.1 Applicare le patch al kernel

Per prima cosa, bisogna localizzare nel sistema la directory contenente i sorgenti del kernel (tipicamente è `/usr/src/linux/`).

Assicurarsi che la release dei sorgenti del kernel sia la *2.4.18*. Per fare questa verifica basta aprire il file chiamato `Makefile` nella suddetta directory, e controllare la presenza delle seguenti linee di comando:

```
VERSION = 2
PATCHLEVEL = 4
SUBLEVEL = 18
```

Se non è presente alcuna directory contenente i sorgenti del kernel, oppure se la release dei sorgenti non è corretta, scaricare una copia del kernel da `http://www.kernel.org`, ed estrarre il contenuto del file in una directory di propria scelta.

Copiare le patch nella directory contenente i sorgenti del kernel, e digitare

```
patch -p1 < nome_della_patch
```

A questo punto è necessario compilare ed installare il kernel.

8.1.2 Compilare ed installare lo scheduler

Se la directory contenente i sorgenti del kernel non è `/usr/src/linux/`, è necessario impostare la variabile di ambiente `KERNEL_DIR` digitando il seguente comando:

```
export KERNEL_DIR =  directory_del_kernel
```

Entrare nella directory contenente il codice dello scheduler (tipicamente è la directory `qres/src/`). Questa directory dovrebbe contenere un file chiamato `Makefile`.

Durante la compilazione è possibile impostare numerose opzioni. Il modo più semplice per compilare il modulo è attraverso il solo comando `make`; in questo modo viene creato il modulo con le caratteristiche di base.

Tuttavia è possibile specificare le seguenti opzioni (le quali vanno inserite dopo il comando `make`):

Opzione	Significato
DEBUG=1	Durante l'esecuzione il modulo stampa alcuni messaggi di warning. Per leggere i messaggi stampati bisogna usare il comando `dmesg`.
MULTI=1	Assegna una banda del 10% ai task di Linux. In questo modo i task del sistema operativo possono andare in esecuzione anche durante l'esecuzione dei task real-time. Inoltre questa opzione permette di assegnare più task ad un singolo server CBS.
PRECISE_ALLOC=1	Abilita l'Hard Reservation: ciascun task real-time esegue esattamente come specificato dalla sua banda (anche se è presente un solo task real-time nel sistema). Come abbiamo visto, questo permette di scavalcare alcuni noti problemi associati all'algoritmo CBS.
GRUB=1	Viene usato l'algoritmo GRUB anzichè l'algoritmo CBS (ricordiamo che GRUB è capace di recuperare la banda inutilizzata).
GRUB_DEBUG=1	Questa opzione funziona come l'opzione DEBUG=1, e va usata soltanto quando è impostata anche l'opzione GRUB=1.

GRUB_EXTRA_BANDWIDTH=1	Quando un task si blocca, la sua capacità residua viene ceduta al successivo task che va in esecuzione.
GRUB_PWR=1	Aggiunge la funzionalità di risparmio energetico allo scheduler. Questa opzione va usata soltanto quando è impostata anche l'opzione GRUB=1. Per il momento è possibile usare questa opzione soltanto su architetture provviste del processore Intel PXA250 (vedi l'opzione 'PXA250').
ARM=1	Abilita un cross-compiling per processori ARM. Richiede che nel sistema siano installate le utility *arm-linux* (i.e. *arm-linux-gcc*).
PXA250=1	Abilita un cross-compiling per processori Intel PXA250. Richiede che nel sistema siano installate le utility *arm-linux* (i.e. *arm-linux-gcc*).

Per installare il modulo nel sistema, digitare `insmod cbs_sched.o`; per rimuovere il modulo dal sistema, digitare `rmmod cbs_sched`. Queste due operazioni richiedono i permessi di root.

Per un corretto funzionamento, si consiglia di compilare il modulo dello scheduler utilizzando la stessa release del compilatore `gcc` che è stata usata per compilare il kernel del sistema operativo. Nel caso si utilizzino versioni differenti, infatti, c'è il rischio che durante il comando `insmod`, Linux si rifiuti di inserire il modulo nel sistema.

Si raccomanda inoltre di *non usare la release 2.96 del compilatore gcc* (utilizzata anche dalla distribuzione *Redhat* 7.3) a causa dei suoi numerosi

bug che potrebbero compromettere il corretto funzionamento dello scheduler real-time.

8.1.3 Compilare i programmi di utilità

Andare nella directory `qres/utils/` e digitare `make` per compilare i programmi *Wrapper* e *Cbser*.

Wrapper è una applicazione che permette di eseguire un task in modalità real-time, assegnandolo ad un server con budget e periodo specificati. Più precisamente, il comando `wrapper` accetta tre parametri: il primo ed il secondo sono, rispettivamente, il budget ed il periodo del server sul quale il nostro programma deve andare in esecuzione, mentre il terzo parametro è il nome del programma da eseguire. La sintassi del comando è quindi la seguente:

```
wrapper Q T COMMAND
```

dove `Q` è il budget del server (espresso in microsecondi), `T` è il periodo del server (espresso in microsecondi), e `COMMAND` è il nome del programma da eseguire (completo di path assoluto, cioè a partire dalla directory /).
Ad esempio, per eseguire il programma `top` con un budget di 200000 e un periodo di 600000 microsecondi, basta digitare

```
./wrapper 200000 600000 /usr/bin/top
```

Cbser è un programma che trasforma un normale task già in esecuzione sul sistema operativo in un task real-time. La sintassi del comando è

```
cbser Q T PID
```

dove `Q` è il budget del server (espresso in microsecondi), `T` è il periodo del server (espresso in microsecondi), e `PID` è il Process Id del task (per conoscere il pid di un determinato task è possibile usare il programma *top*). Ad esempio, se il programma *find* sta girando con pid pari a 2001, e lo si vuole eseguire su un server con un budget di 200000 microsecondi ed un periodo di 600000 microsecondi, basta digitare

```
./cbser 200000 600000 2001
```

Andare nella directory `qres/tests/` e digitare `make`; questo comando crea due programmi chiamati *Schedtest* e *Filter*.

Schedtest è un programma che può essere usato per testare lo scheduler real-time. La sintassi del comando è

```
schedtest Q T D
```

dove `Q` è il budget del server (espresso in microsecondi), `T` è il periodo del server (espresso in microsecondi), e `D` indica dopo quanti secondi interrompere il test (infatti, dato che non è possibile conoscere a priori la durata complessiva del test, è desiderabile poter impostare una durata massima).
Se si vuole compilare il programma schedtest per le architetture con processori i386 o PowerPC è necessario modificare, rispettivamente, i file `qres/include/i386-rdtsc.h` e `qres/include/ppc-rdtsc.h`, impostando la variabile `SCALE` uguale alla frequenza della CPU che si sta utilizzando (espressa in KHz). Per conoscere il corretto valore della variabile basta digitare in una shell il seguente comando

```
cat /proc/cpuinfo
```

leggere il valore della variabile `cpu MHz`, e moltiplicarlo per 1000.

Il programma *Filter* estrae alcune informazioni dall'output prodotto dal programma schedtest, e crea una traccia grafica visualizzabile con il programma *xfig*.

Ad esempio, per eseguire due test, con budget di 200000 e 400000 microsecondi, ed un periodo di 600000 microsecondi, per una durata di 30 secondi, basta digitare

```
./schedtest 200000 600000 30 > a &
./schedtest 400000 600000 30 > b
```

Questo comando esegue due test, e crea due file di output, chiamati `a` e `b`. Per vedere la schedulazione ottenuta in modo grafico, , digitare

```
./filter a b > c.fig
```

Questo comando crea un file chiamato `c.fig` che può essere visualizzato utilizzando l'applicazione *xfig*.

8.2 Test

Descriviamo ora la serie di test che abbiamo effettuato sulla macchina Intrinsyc CerfCube per verificare il corretto funzionamento dello scheduler real-time. In particolare, ciascun test è stato fatto con lo scopo di rilevare eventuali errori in una determinata porzione del codice. É stato poi fatto un test conclusivo, che utilizza tutte le funzionalità offerte dal nostro scheduler (algoritmo GRUB, più task gestiti da un unico server, risparmio energetico).

8.2.1 Test dell'algoritmo

I primi test sulla macchina sono stati effettuati per verificare il corretto funzionamento dell'algoritmo di schedulazione (GRUB), ed in particolare per controllare il codice relativo alla lettura del timer (registro OSCR) e relativo alle conversioni di tempo secondo le diverse unità di misura già viste.

É stata utilizzata l'applicazione `schedtest` già descritta nei precedenti paragrafi. Abbiamo provato il sistema eseguendo due `schedtest` contemporaneamente, il primo con banda 200000 e periodo 600000, ed il secondo con banda 300000 e periodo 900000:

```
schedtest 200000 600000 30 > a & schedtest 300000 900000 30 > b
```

Come si può vedere, abbiamo rediretto l'output su due file (a e b). Questi file sono stati poi elaborati attraverso l'applicazione `filter`, la quale ha prodotto un file leggibile attraverso l'applicazione `xfig`. Il risultato, ossia la schedulazione ottenuta, è mostrata in Figura 8.1.

Procedendo allo stesso modo, abbiamo eseguito altri due `schedtest`, eseguendo un task con banda 360000 e periodo 900000, ed un altro task con banda 120000 e periodo 600000:

```
schedtest 360000 900000 30 > a & schedtest 120000 600000 30 > b
```

La schedulazione che abbiamo ottenuto è mostrata in Figura 8.2.

8.2.2 Test del cambio di frequenza

Verificato il corretto funzionamento dello scheduler, siamo passati alla verifica del funzionamento del codice relativo al cambio di frequenza del processore. Volendo trovare un metodo veloce per fare il controllo, abbiamo deciso di mandare in esecuzione un task semplicissimo, che si limita a contare fino all'infinito (attraverso un ciclo `for`) ed a stampare periodicamente il valore del contatore. Il codice che abbiamo utilizzato è il seguente:

```
#include <stdlib.h>

int main()
{
  long int a = 0;
  for (;;) {
     a++;
     if ((a % 10000)==0) {
        printf("%ld\n",a);
     }
  }
}
```

Si noti che questo semplice codice non ha bisogno di accessi in memoria (il valore delle variabili viene mantenuto all'interno dei registri del processore), perciò la velocità con cui viene eseguito è direttamente proporzionale alla frequenza di clock del processore.

Poichè la macchina Intrinsyc CerfCube è sprovvista di output, abbiamo deciso di leggere il valore del contatore attraverso una connessione con `SSH`. Tuttavia, per permettere al CerfCube di inviare i dati in rete, è stato necessario eseguire i task del sistema operativo in modalità real-time (altrimenti la parte del sistema operativo dedicata alla connessione SSH sarebbe andata in stallo, e non ci sarebbe stato alcun output). Abbiamo quindi assegnato al sistema operativo una banda $U_L = 0.1$ per eseguire i propri task.

Una volta compilato il suddetto codice, lo abbiamo mandato in esecuzione con l'applicazione `wrapper`. Abbiamo eseguito l'applicazione diverse volte, ed

ogni esecuzione è durata un minuto. Ricordiamo che variare l'utilizzazione totale del sistema, significa variare la frequenza di clock del processore PXA250, cioè variare la potenza di calcolo disponibile. I risultati che abbiamo ottenuto sono riassunti nella seguente tabella:

Presenza del modulo	Banda del task	Banda totale del sistema	Frequenza della CPU	Ultimo valore stampato
NO	-	-	100 MHz	35580000
SI	0.149	0.249	100 MHz	33190000
SI	0.399	0.499	200 MHz	64540000
SI	0.899	0.999	400 MHz	122990000

Come si vede, all'aumentare del carico del sistema, il numero di cicli effettuati dalla nostra applicazione è cresciuto in modo proporzionale alla frequenza di clock del processore. Questo dimostra che *la frequenza di clock è effettivamente cambiata.*

Come avevamo previsto, la crescita è stata praticamente lineare, perché l'applicazione non ha effettuato accessi in memoria.

8.2.3 Test del Virtual Time

Nel terzo capitolo abbiamo visto che il valore del virtual time V_i in ogni istante è una misura di quanto servizio riservato al server S_i è stato consumato fino a quell'istante di tempo. Abbiamo anche visto che l'algoritmo GRUB cerca di aggiornare il valore di V_i in modo che, ad ogni istante di tempo, il server S_i abbia ricevuto la stessa quantità di servizio che avrebbe ricevuto in un tempo V_i se fosse stato eseguito su un processore dedicato con capacità U_i.

Abbiamo controllato la correttezza di questa affermazione all'interno del nostro sistema real-time. Dedicando a Linux una banda $U_L = 0.1$, abbiamo mandato in esecuzione il task del precedente test con una banda pari a $U_1 = 0.149$. Per far si che Linux esaurisse tutta la banda a disposizione, abbiamo inserito tra i suoi task un'applicazione che esauriva tutto il tempo di CPU che gli veniva concesso; per creare questo task abbiamo utilizzato un semplice ciclo infinito:

```
int main()
```

```
{
  for (;;) ;
}
```

In questo modo abbiamo avuto la certezza che nessuna porzione della banda di Linux venisse ceduta al task di conteggio.

Il nostro task, perciò, è andato in esecuzione sul processore a frequenza 100 MHz con una banda pari a 0.149/0.249 della banda totale del sistema; secondo l'affermazione precedente, questo equivale ad aver eseguito il task su un sistema con un processore a (0.149/0.249)*100MHz = 60 MHz. Tramite l'applicazione `wrapper`, abbiamo eseguito il task di conteggio per un minuto, ottendendo il seguente risultato:

Presenza del modulo	Banda del nostro task	Banda totale del sistema	Frequenza della CPU	Ultimo valore stampato
SI	0.149	0.249	100 MHz	25080000

Sempre nelle stesse condizioni, abbiamo mandato in esecuzione un ulteriore task che utilizzava pienamente una banda $U_2 = 0.750$ (in pratica, è stata mandata in esecuzione un'altra istanza del ciclo infinito appena descritto). Questa volta il task di conteggio è andato in esecuzione sul processore a frequenza 400 MHz con una banda pari a 0.149/0.999 della banda totale del sistema: anche in questo caso è come se il task fosse andato in esecuzione su un sistema con processore a (0.149/0.999)*400MHz = 60 MHz. Anche questa volta abbiamo eseguito il task per un minuto, aspettandoci di ottenere un valore vicino a quello ottenuto nel precedente caso. Il risultato che abbiamo ottenuto è il seguente:

Presenza del modulo	Banda del nostro task	Banda totale del sistema	Frequenza della CPU	Ultimo valore stampato
SI	0.149	0.999	400 MHz	25090000

Questo risultato è effettivamente molto vicino a quello previsto teoricamente.

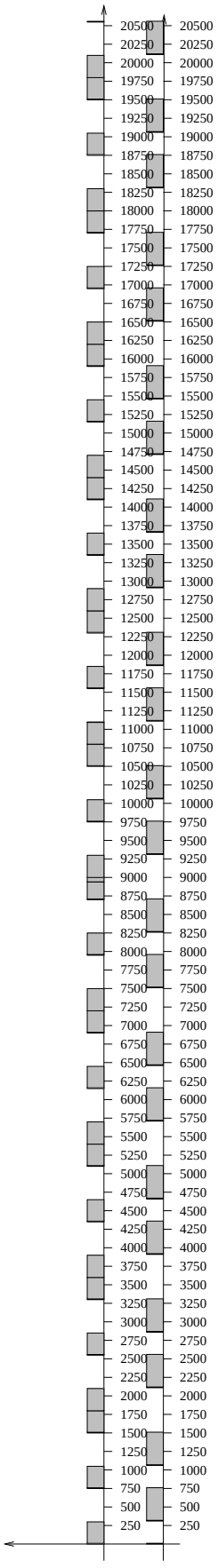

Figura 8.1: Primo test

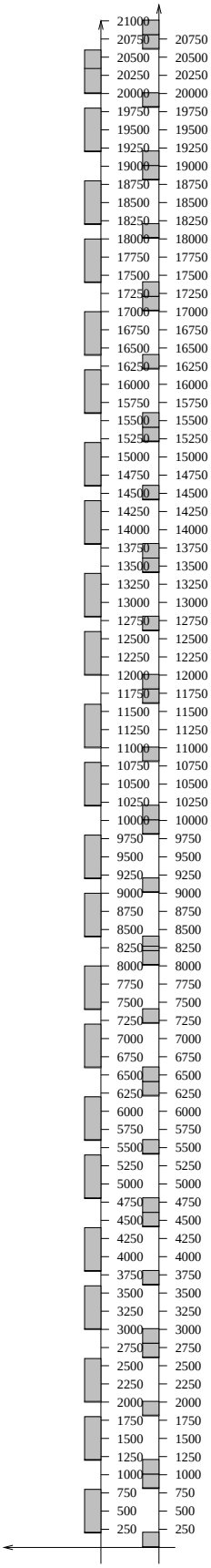

Figura 8.2: Secondo test

8.2.4 Test per applicazioni multimediali

Si potrebbe pensare che la variazione della frequenza di clock della CPU mantenendo invariata la velocità delle periferiche (come ad esempio la memoria) non comporti un apprezzabile variazione nelle prestazioni del sistema.

Dato che il nostro studio è particolarmente rivolto alle applicazioni di tipo multimediale, abbiamo deciso di valutare l'andamento delle prestazioni di un'applicazione di questo tipo al variare della frequenza di clock della sola CPU.

Poichè la macchina Intrinsyc CerfCube è sprovvista di video, abbiamo deciso di utilizzare un programma di decompressione per file audio. La nostra scelta è caduta sul decoder fornito dalla *Xiph.Org Foundation*, che è una associazione senza scopo di lucro dedita a proteggere da interessi privati le basi multimediali di Internet; lo scopo di questa associazione è quello di promuovere e sviluppare protocolli e software open source e disponibili a tutti.

Ogg Vorbis è un formato audio compresso non proprietario di alta qualità (da 8 a 48 kHz, 16 bit, polifonico) con bitrate fisso o variabile da 16 a 128 Kbps per canale; queste specifiche permettono al formato di essere in diretta competizione con le altre codifiche audio in commercio, come MPEG-4 e simili.

In particolare, *Tremor* è una libreria di decodifica completamente compatibile con il formato audio Ogg Vorbis, e che utilizza esclusivamente l'aritmetica intera. Questa caratteristica si presta molto bene al nostro caso, dato che la macchina Intrinsyc CerfCube è sprovvista di unità FPU (Floating Point Unit). La libreria Tremor è scaricabile via CVS con i seguenti comandi:

```
export CVSROOT=:pserver:anoncvs@xiph.org:/usr/local/cvsroot
cvs login
cvs co Tremor
```

utilizzando la password `anoncvs`.

Per eseguire il test abbiamo decompresso alcuni stream audio a 44100 Hz e 2 canali, utilizzando l'applicazione `ivorbisfile_example`, distribuita con

la suddetta libreria.
Attraverso il comando `time` abbiamo misurato il tempo necessario alla macchina per decodificare ogni stream alle diverse frequenze di clock della CPU. Lo stream decodificato è stato rediretto verso `/dev/null`. La linea di comando completa è:

```
time ./wrapper Q T /bin/ivorbisfile_example < ./file.ogg > /dev/null
```

La misura del tempo necessario alla decodifica è stata effettuata per ogni stream audio e per ogni frequenza di clock della CPU. I risultati ottenuti sono riassunti nella seguente tabella:

Dimensione stream audio (bytes)	Bit Rate (kbps)	Numero campioni	Tempo a 100 MHz (secondi)	Tempo a 200 MHz (secondi)	Tempo a 400 MHz (secondi)
265009	69	1323001	28.460	14.529	7.939
1003199	128	2876477	63.712	32.415	17.658
1052340	128	3145721	69.431	35.278	19.200
141509	45	1323000	31.162	16.050	8.918
5816848	116	17640000	391.616	198.456	107.466
234750	62	1323001	28.064	14.383	7.852
3900576	117	11667456	257.128	130.357	70.624

Da questi valori abbiamo estratto la crescita della velocità di decompressione all'aumentare della frequenza di clock della CPU; supponendo unitaria la velocità di decompressione di ciascuno stream a 100 MHz, abbiamo:

Esempio	Velocità a 200 MHz	Velocità a 400 MHz
1	1.959	3.585
2	1.965	3.608
3	1.968	3.616
4	1.941	3.494
5	1.973	3.644
6	1.951	3.574
7	1.972	3.640

Abbiamo quindi calcolato la media dei campioni, e l'intervallo di confidenza al 95% e al 99% Un intervallo di confidenza è un range di valori che include, con una specifica probabilità, il parametro che si vuole stimare. Per l'intervallo di confidenza è stata usata la seguente formula:

$$\mu = m \pm t_{\alpha/2}^{n-1} \sqrt{\frac{\sum_{i=1}^{n}(X_i - m)^2}{(n-1)n}}$$

dove X_i è l'i-esimo campione, m è la media dei campioni, n è il numero dei campioni, e $t_{\alpha/2}^{n-1}$ è il valore della variabile standardizzata che esclude il α% ($\frac{\alpha}{2}$ per parte) della distribuzione di t definita da (n-1) gradi di libertà della varianza stimata.

Gli intervalli di confidenza della velocità di decompressione rispetto alle diverse frequenze di clock della CPU sono raccolti nella seguente tabella:

Livello di confidenza	Velocità a 200 MHz	Velocità a 400 MHz
95%	1.961 ± 0.011	3.594 ± 0.047
99%	1.961 ± 0.016	3.594 ± 0.072

L'andamento della velocità di decompressione in funzione della frequenza di clock della CPU è mostrato in Figura 8.3, nella quale è indicato anche l'intervallo di confidenza al 99%:

8.3 Risparmio energetico

Abbiamo infine valutato la potenza consumata dal dispositivo alle diverse frequenze di clock della CPU.

Dato che la tensione in ingresso al CerfCube è sempre di 5 Volt in continua, per calcolare la potenza assorbita dal dispositivo è stato sufficiente misurare la corrente in ingresso ad esso. Abbiamo perciò realizzato un circuito alimentato separatamente (mediante una batteria da 9 Volt), che provvede a misurare la corrente in ingresso al CerfCube e ad inviarne la misura (opportunamente amplificata) ad un computer incaricato della raccolta dei dati. Il circuito inserisce una piccola resistenza in serie al CerfCube, e misura la

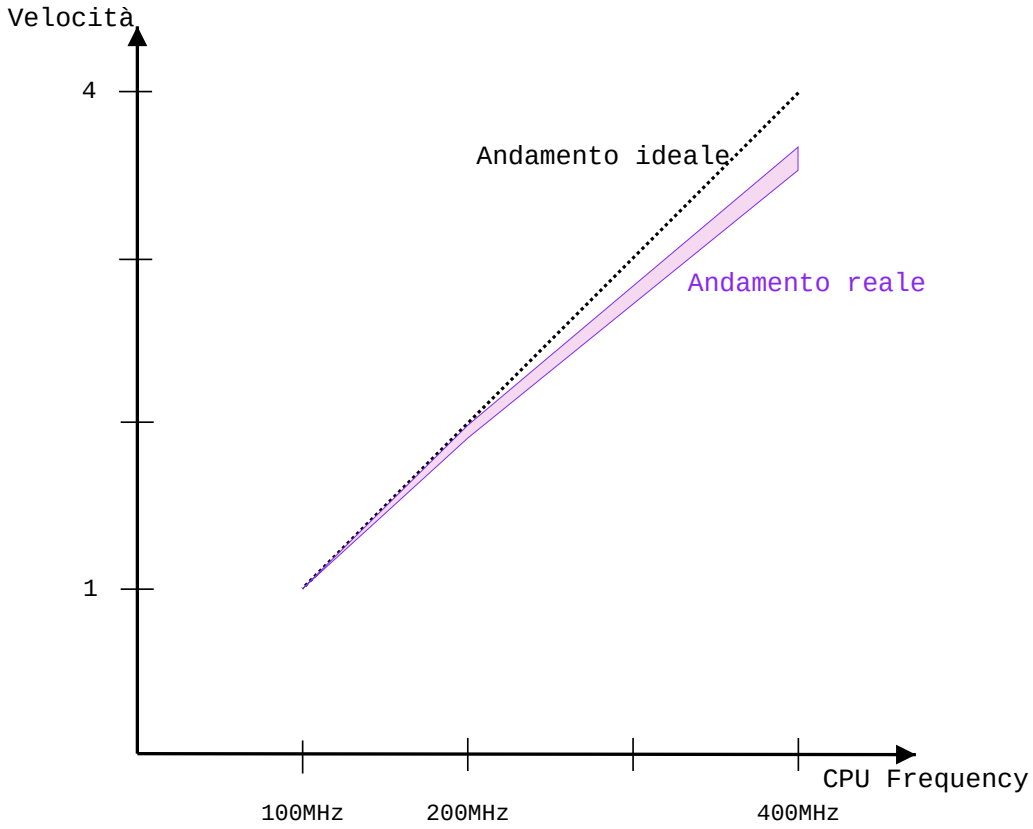

Figura 8.3: Velocità di decompressione in funzione della frequenza di clock (confidenza al 99%)

tensione ai suoi capi; poiché il valore della resistenza è noto, il passaggio dalla misura della tensione al valore dell'intensità di corrente in ingresso al dispositivo è immediato.

Per la trasmissione dei dati è stato usato un collegamento seriale a *57600 bps* con 8 bit di dati, nessuna parità, 1 bit di stop (8N1) e nessun controllo di flusso.

8.3.1 Calcolo del consumo

Il misuratore di corrente esprime il valore misurato mediante un valore su 8 bit (cioè su una scala di valori da 0 a 255). La taratura (cioè l'associazione del valore ricevuto attraverso la seriale al corrispondente valore dell'intensità di corrente) è stata fatta mediante un tester. La corrispondenza è mostrata nella seguente tabella:

Frequenza di clock della CPU	Stato del sistema operativo	Valore ricevuto via seriale	Corrente misurata
100 MHz	idle	8	270 mA
400 MHz	carico	231	590 mA

Per caricare il sistema è stato eseguito il programma `ivorbisfile_example` precedentemente descritto; si noti che questa applicazione fa uso della CPU, della memoria e della Flash ROM (dalla quale legge lo stream da decodificare). All'applicazione è stata riservata una banda pari a 0.9, mentre gli altri task del sistema operativo sono stati eseguiti su un unico server GRUB con banda 0.1.

Da questi dati possiamo ricavare la relazione che lega il valore dell'intensità di corrente al valore letto attraverso la seriale. La relazione è la seguente:

$$i = 258.52mA + valore * 1.435mA$$

Avendo a disposizione un circuito per misurare la potenza assorbita dalla macchina, abbiamo studiato l'andamento del consumo alle diverse frequenze di clock. Per effettuare questo studio abbiamo eseguito l'applicazione `ivorbisfile_example` mediante il comando `wrapper`, specificando diversi valori della banda (si ricordi che la frequenza di clock della CPU è proporzionale alla banda del sistema). Durante tutte le misurazioni, i task del sistema operativo sono stati eseguiti su un unico server GRUB con banda 0.1. Dai dati ricevuti tramite la seriale, abbiamo poi estratto[1] l'andamento temporale

[1]Per l'estrazione è stata usata l'applicazione *gnuplot*.

dell'assorbimento di corrente, che abbiamo riportato nelle figure 8.4, 8.5 e 8.6.

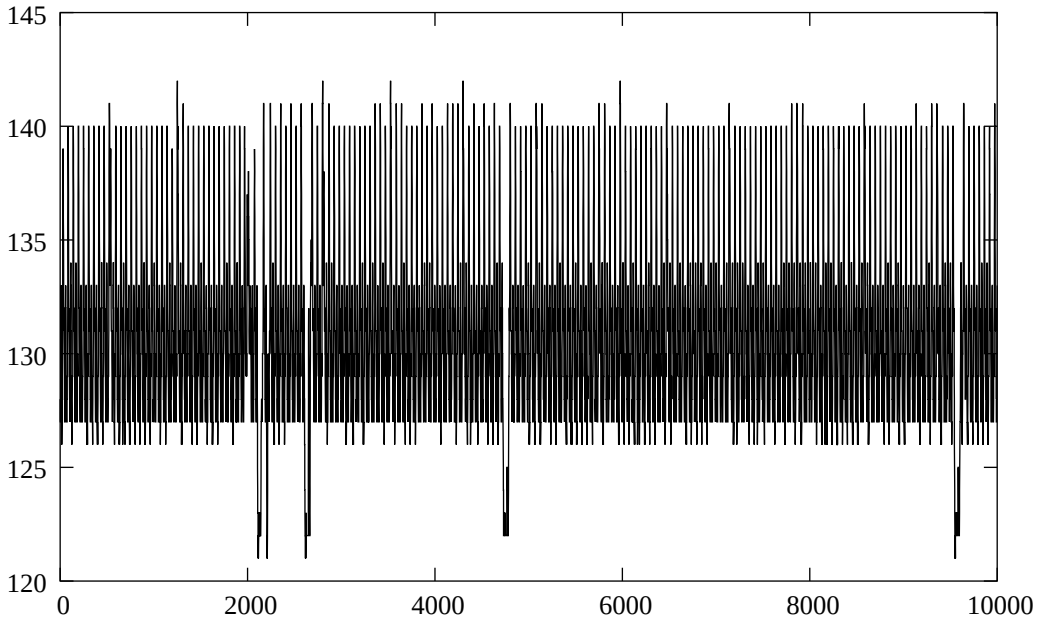

Figura 8.4: Assorbimento quando U = 0.25

In Figura 8.7 è inoltre riportato il grafico dell'assorbimento di corrente che si ottiene caricando istantaneamente il sistema operativo (mandando, cioè, in esecuzione l'applicazione `ivorbisfile_example` con banda 0.9 quando il sistema era idle).

Abbiamo inoltre calcolato il valor medio dell'assorbimento alle diverse frequenze di clock, e lo abbiamo riportato nella seguente tabella:

Frequenza di clock della CPU	Stato del sistema operativo	Valor medio letto via seriale	Corrente assorbita
100 MHz	carico	130.653	446.0 mA
200 MHz	carico	174.204	508.5 mA
400 MHz	carico	223.943	579.9 mA

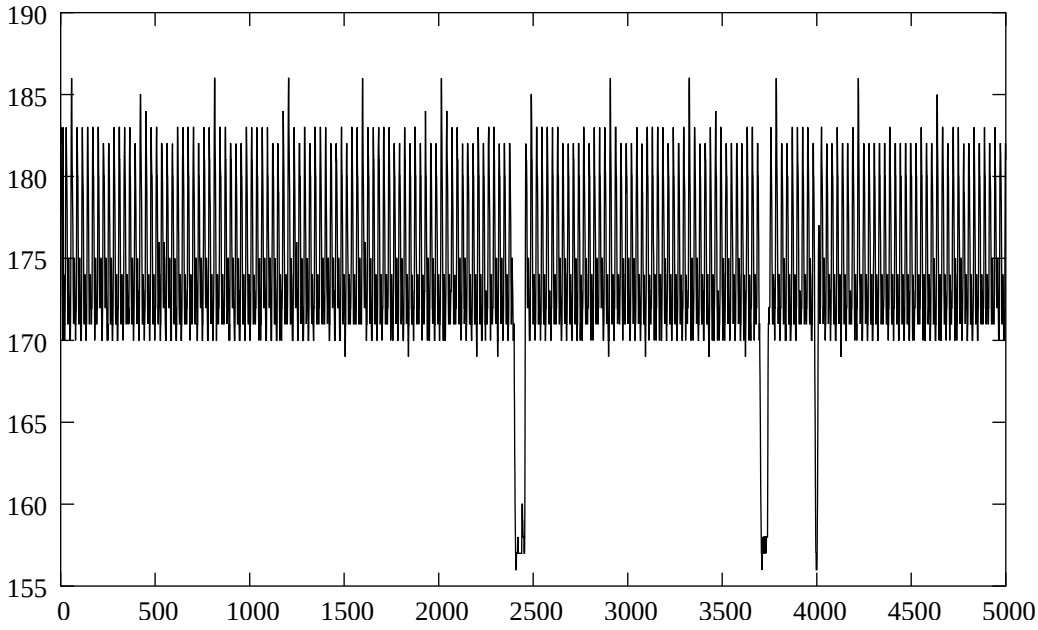

Figura 8.5: Assorbimento quando U = 0.5

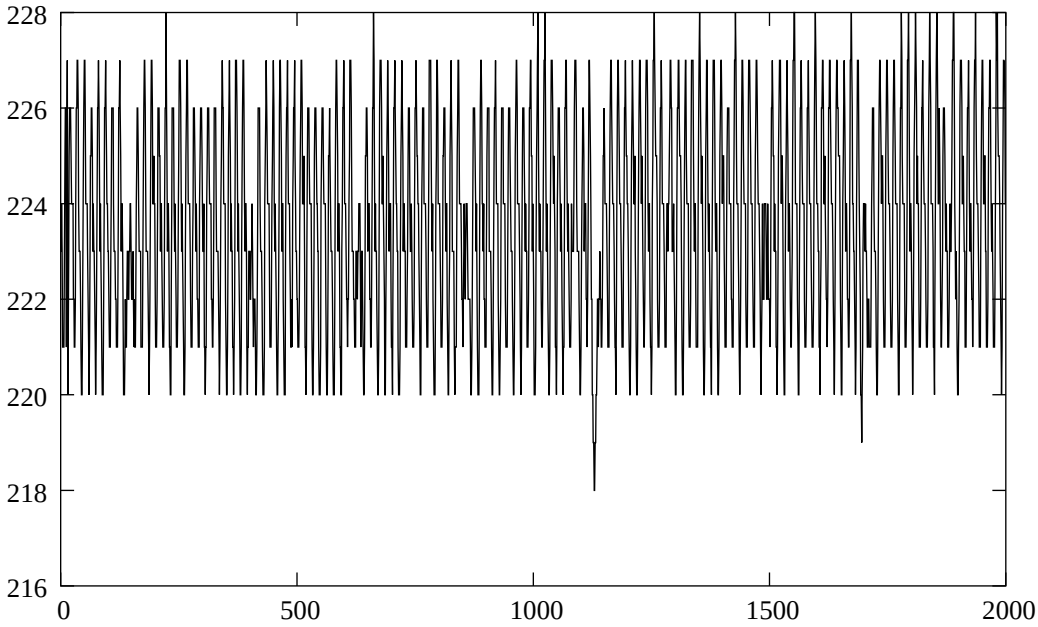

Figura 8.6: Assorbimento quando U = 1

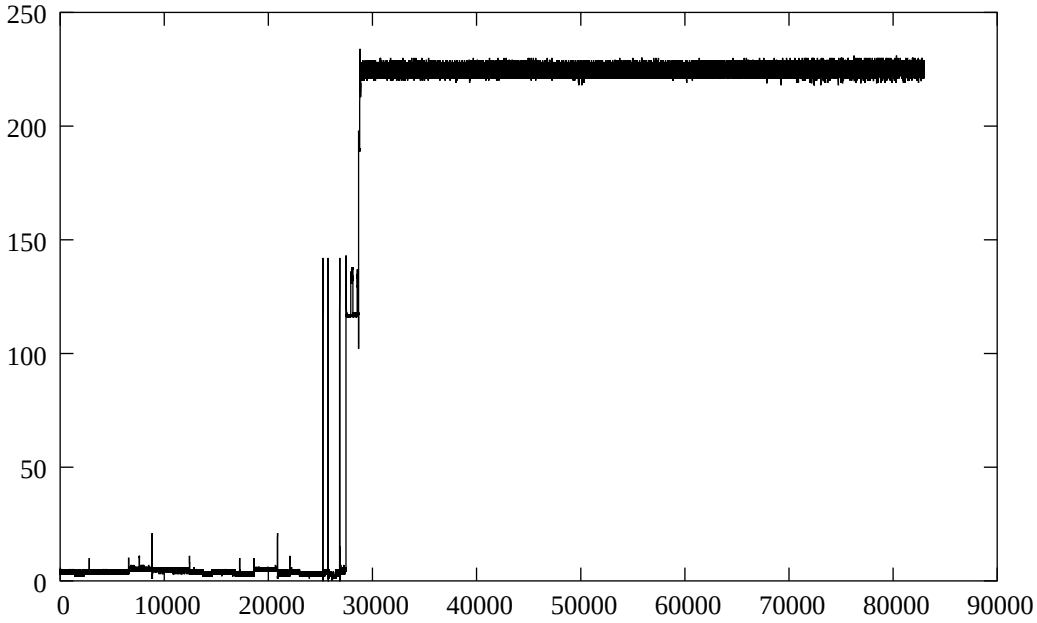

Figura 8.7: Passaggio da U = 0.1 a U = 1

8.3.2 Valutazione del risparmio energetico

I benefici apportati dall'utilizzo del nostro scheduler si presentano in due situazioni distinte:

1. Quando il sistema è idle, è possibile abbassare la frequenza di clock a 100 MHz. Senza il meccanismo di voltage scaling, invece, è necessario mantenere la frequenza fissa a 400 MHz, per poter garantire le massime prestazioni durante i picchi di carico.

   L'assorbimento di corrente che si misura con il voltage scaling quando il sistema è idle è mostrato in Figura 8.8.

   Il valor medio della corrente assorbita in questo caso risulta essere 1.346 (cioè 250.5 mA).

   Nel caso in cui non si disponga dell'algoritmo di voltage scaling, è necessario mantenere il sistema alla frequenza massima (400 MHz) anche nei momenti in cui il sistema è idle, per non avere perdite di performance

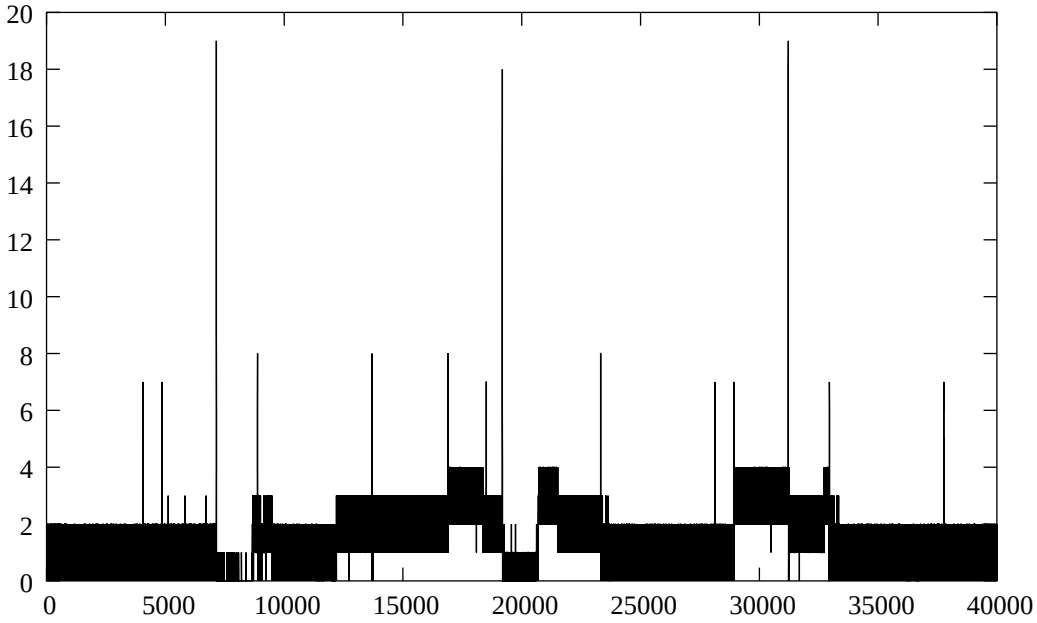

Figura 8.8: Sistema idle con voltage scaling

durante i picchi. L'assorbimento di corrente in questo caso è mostrato nella Figura 8.9.

Il valor medio della corrente assorbita dal dispositivo in questo caso risulta essere 103.313 (cioè 406.8 mA). In altre parole, *l'utilizzo del voltage scaling permette di risparmiare il 38% della potenza assorbita tutte le volte che il sistema è idle.*

2. Quando il sistema non è idle, e la banda totale del sistema non è unitaria, è possibile trovare una frequenza di clock inferiore a 400 MHz che permetta di rispettare le deadline dei task real-time. Ad esempio, se la banda totale del sistema è pari a 0.5, è possibile utilizzare una frequenza di clock di 200 MHz, anzichè essere costretti ad utilizzarne una di 400 MHz.

   Per studiare questo caso abbiamo modificato il programma `ivorbis file_example` creando due thread (è stata utilizzata la libreria *pthread*) che condividono un buffer di dimensione arbitraria; il primo thread decodifica lo stream audio e scrive nel buffer, mentre il secondo

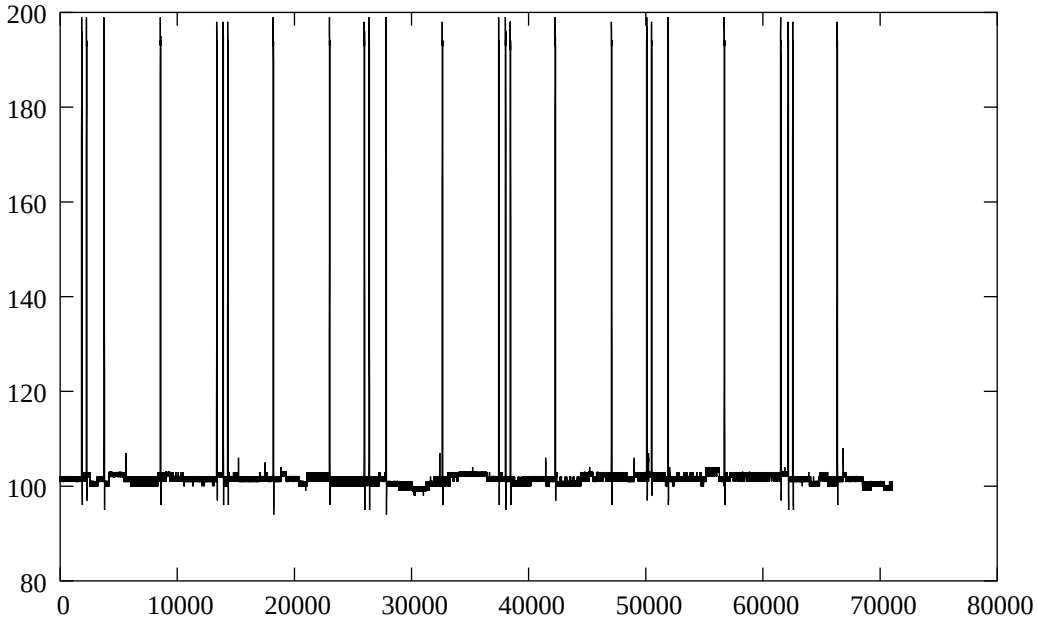

Figura 8.9: Sistema idle senza voltage scaling

thread legge dal buffer e ne fa un monitoraggio periodico. Il tempo di esecuzione dell'applicazione così modificata non risulta più essere sensibile alla frequenza di clock della CPU, ma è legato esclusivamente alla durata dello stream audio.

Abbiamo fatto in modo che ciascuna istanza dell'applicazione non resti inattiva per un tempo maggiore ad un secondo (in pratica, abbiamo impostato il periodo con cui viene monitorato il buffer esattamente ad un secondo). In questo modo ciascun task real-time non è mai nello stato `Inactive` per un tempo sufficiente a causare una diminuzione della frequenza di clock della CPU (si ricordi che la variabile `PWR_TIMEOUT` è stata impostata a tre secondi).

Abbiamo eseguito l'applicazione modificata decodificando uno stream audio con una banda di 0.15. In presenza di voltage scaling abbiamo misurato un assorbimento medio di 59.296 (equivalente a 343.6 mA). Senza voltage scaling l'assorbimento medio è stato di 121.807 (pari a 433.3 mA). *Il risparmio di energia è stato perciò del 20.7 %.*

Volendo studiare un caso più generale, abbiamo creato un test nel quale il sistema operativo esegue più istanze dell'applicazione modificata, decodificando contemporaneamente differenti stream audio. A ciascuna istanza è stata assegnata una banda diversa attraverso il comando `wrapper`, in modo da ottenere una banda del sistema con un valore variabile nel tempo e compreso tra 0 e 1.
Lo script del test è il seguente:

```
echo Esecuzione in background di uno stream lungo 264 secondi:
./wrapper 149000 1000000 /home/cloud/vorbis13 < oggs/264.ogg > /dev/null &
sleep 30
echo Esecuzione in background di uno stream lungo 30 secondi:
./wrapper 500000 1000000 /home/cloud/vorbis13 < oggs/30.ogg > /dev/null &
echo Esecuzione di uno stream lungo 65 secondi:
./wrapper 250000 1000000 /home/cloud/vorbis13 < oggs/65.ogg > /dev/null
sleep 30
echo Esecuzione in background di uno stream lungo 71 secondi:
./wrapper 250000 1000000 /home/cloud/vorbis13 < oggs/71.ogg > /dev/null &
sleep 41
echo Esecuzione di uno stream lungo 30 secondi:
./wrapper 500000 1000000 /home/cloud/vorbis13 < oggs/30.ogg > /dev/null
```

L'andamento della banda che otteniamo con questo test è mostrato in Figura 8.10. A questo andamento dovrebbe corrispondere un andamento molto simile dell'assorbimento di corrente da parte del dispositivo.

L'assorbimento di corrente misurato in presenza del nostro algoritmo è mostrato in Figura 8.11, ed ha effettivamente un andamento molto simile a quello mostrato in Figura 8.10.

Il valor medio ricevuto via seriale è stato 107.864 (corrispondente a 413.3 mA).

L'assorbimento di corrente eseguendo lo stesso test sul dispositivo con frequenza di clock fissa a 400 MHz è mostrato in Figura 8.12.

Il valor medio misurato è stato 141.219 (corrispondente a 461.2 mA). *Il risparmio di energia che abbiamo ottenuto è stato perciò del 10.4 %.*

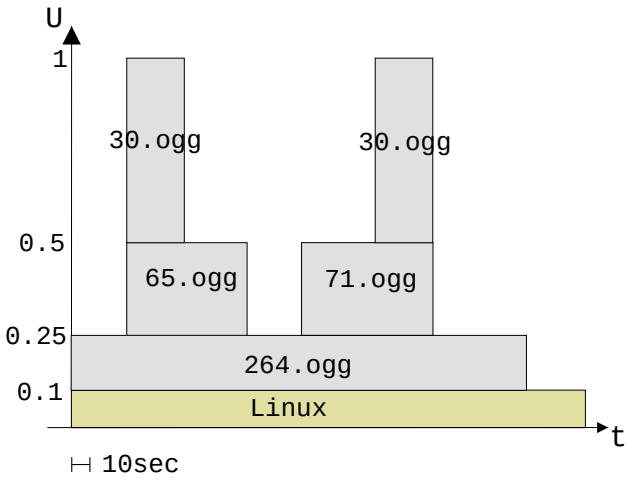

Figura 8.10: Variazione della banda durante il test

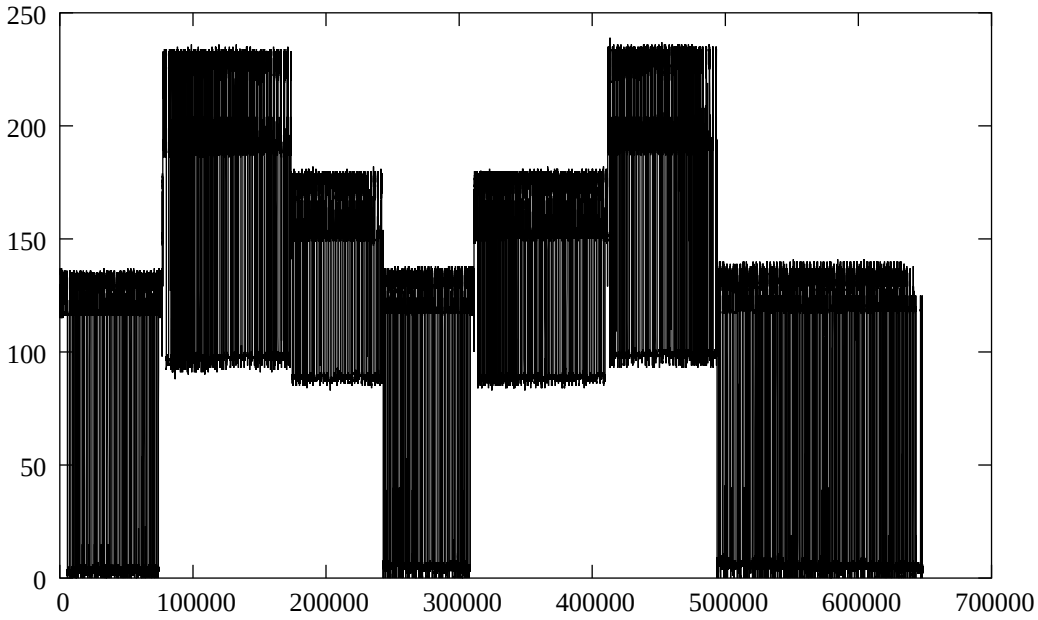

Figura 8.11: Sistema carico con voltage scaling

8.4 Conclusioni e lavoro futuro

Abbiamo realizzato un sistema operativo general purpose che abbina le garanzie di un sistema soft real-time alle capacità di risparmio energetico. Il

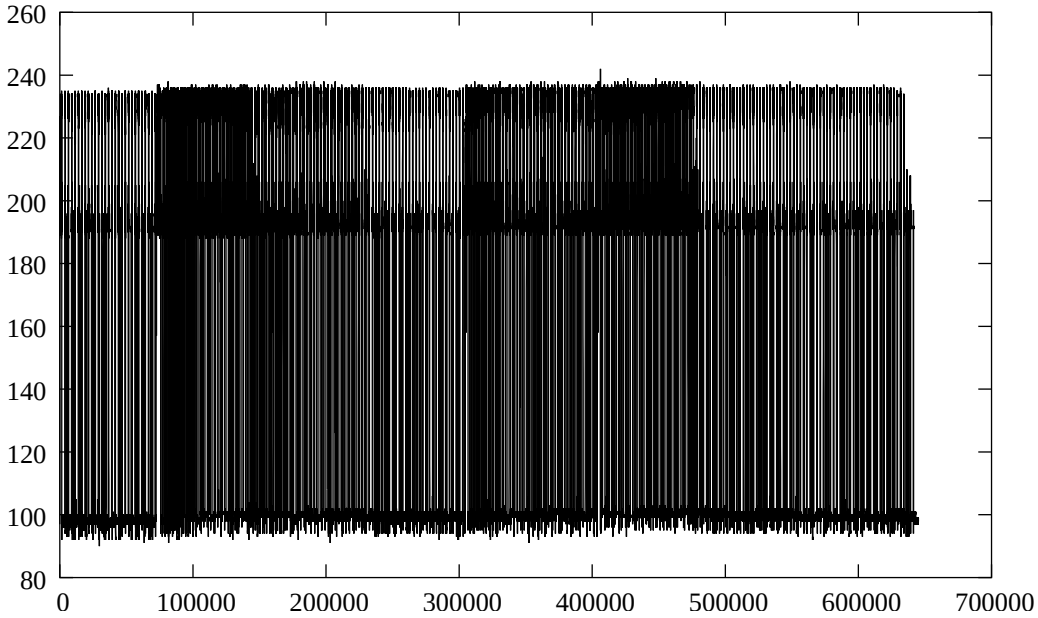

Figura 8.12: Sistema carico senza voltage scaling

sistema è altamente modulare, open source (sviluppato sotto licenza GPL[2]) e supporta le architetture con processore Intel PXA250.

Tra i possibili sviluppi futuri dello scheduler citiamo i seguenti:

- Ottimizzazione del codice (in modo da ridurre ulteriormente l'overhead introdotto dallo scheduler).
- Implementazione di nuovi algoritmi di schedulazione.
- Ampliamento delle architetture supportate (per le architetture con processori i386, PowerPC e Strong Arm 1110 rimane soltanto da scrivere il codice per il risparmio energetico).

[2]La GPL, o *General Public License*, è stata creata dalla comunità di sviluppatori GNU, e permette di modificare codice Open Source a patto di distribuire il codice sorgente modificato. La licenza è disponibile al seguente URL: *http://www.gnu.org/licenses/gpl.html*

Bibliografia

[Abe98] L.Abeni and G.Buttazzo, *Integrating Multimedia Applications in Hard Real-Time Systems*, Proceedings of the IEEE Real-Time Systems Symposium, Madrid, Spain, pp. 4-13, December 1998. IEEE Computer Society Press.

[Abe02] Luca Abeni and Giuseppe Lipari, *Implementing Resource Reservations in Linux*, Real-Time Linux Workshop, Boston (MA), December 2002.

[Bin01] Enrico Bini, Giorgio Buttazzo, Giuseppe Buttazzo, *A Hyperbolic Bound for the Rate Monotonic Algorithm*, Proceedings of the Euromicro Conference on Real-Time Systems, Delft, The Netherlands, June 2001.

[But] Giorgio C.Buttazzo, *Sistemi in Tempo Reale*, Pitagora Editrice Bologna.

[But99] G.Buttazzo and F.Sensini, *Optimal Deadline Assignment for Scheduling Soft Aperiodic Tasks in Hard Real-Time Environments*, IEEE Transactions on Computers, Vol. 48, No. 10, October 1999.

[Cof73] Jr.E.Coffman and P.J.Denning, *Operating Systems Theory*, Prentice-Hall, Englewood Cliffs, NJ, 1973.

[Gha95] T.M.Ghazalie and T.P.Baker, *Aperiodic Servers In A Deadline Scheduling Environment*, The Journal of Real-Time Systems, 9, 1995.

[Intel02] Intel corporation, *Intel PXA250 and PXA210 Application Processors Developer's Manual*, order number 278522-001, February 2002.

[Lip00] Giuseppe Lipari and Sanjoy Baruah, *Greedy reclamation of unused bandwidth in constant-bandwidth servers*, Proceedings of the EuroMicro Conference on Real-Time Systems, pp. 193-200, Stockholm, Sweden. June 2000. IEEE Computer Society Press.

[Lip01] Giuseppe Lipari and Sanjoy Baruah, *A hierarchical extension to the constant bandwidth server framework*, Proceedings of the Real-Time Technology and Applications Symposium, Taipei, Taiwan. May 2001. IEEE Computer Society Press.

[Liu73] C.L.Liu and J.W.Layland, *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*, Journal of the ACM, Vol. 20, No. 1, January 1973.

[Mar] Franco Martorana, *Istituzioni di metodologia statistica*, SEU.

[Oce02] Ismael Ripoll, Pavel Pisa, Luca Abeni, Paolo Gai, Agnes Lanusse, Sergio Saez, and Bruno Privat, *WP1 - RTOS State of the Art Analysis: Deliverable D1.1 - RTOS Analysis*, Ocera 2002.

[Raj98] R. RajKumar, K. Juvva, A. Molano, and S. Oikawa, *Resource kernels: A resource-centric approach to real-time systems*, Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking, January 1998.

[RTLinux] Victor Yodaiken, *The RTLinux Manifesto*, Department of Computer Science - New Mexico Institute of Technology.

[Rub] Alessandro Rubini and Jonathan Corbet, *Linux Device Drivers 2nd edition*, O'REILLY.

[Spu94] M.Spuri and G.C.Buttazzo, *Efficient Aperiodic Service under Earliest Deadline Scheduling*, Proceedings of the 15th IEEE Real-Time Systems Symposium, San Juan, Puerto Rico, 1994.

[Spu96] M.Spuri and G.C.Buttazzo, *Scheduling Aperiodic Tasks in Dynamic Priority Systems* , The Journal of Real-Time Systems, 10, 1996.

Ringraziamenti

Desidero ringraziare i professori Paolo Ancilotti e Giuseppe Lipari, per avermi dato la possibilità di svolgere la tesi presso il ReTiS Lab della Scuola Superiore Sant'Anna.

Ringrazio anche tutti i ragazzi del laboratorio, ed in particolare Luca Abeni per l'enorme disponibilità che ha sempre mostrato nei miei confronti.

Grazie anche a Franco Zaccone che ha realizzato il misuratore di potenza presso il laboratorio ARTS.

Infine, un sincero ringraziamento a Dio per avermi dato la vita, alla mia famiglia per avermi dato la possibilità di studiare, e a Silvia per avermi sopportato durante i periodi di esame.