Sistemi Embedded

RIVA 'ZAX' SAMUELE
copyleft - all rights reVerSed

Sommario

- **Introduzione**
- **HW-SW codesign**
    - Flusso di progettazione
- **Metriche**
    - Cost Modeling
- **Tecnologie**
    - Tipologie di processori: general purpose, applicatio specific...
    - Esecutori Hardware: application specific IC, sistemi riconfigurabili...
    - Architetture di comunicazione: bus, interfacce wireless...
- **RTOS**
    - Sistemi operativi Real-Time: caratteristiche, scheduling, configurazione...
    - Analisi del software embedded: strumenti di analisi, scrittura di driver...
    - Comunicazione: bus e porte.
- **Resistenza ai guasti (fault tolerance)**
    - Hardware fault tolerant
    - Software fault tolerant
    - Il caso dei sistemi operativi
    - Basi di dati
    - Reti
    - Esempi applicativi

Introduzione

La gran parte dei computer esistenti sono sistemi embedded, inseriti in cellulari, lettori mp3, fotocamere, automobili, elettrodomestici... ormai quasi tutto quello che funziona tramite elettricità è spesso regolato da un sistema computerizzato embedded!

Questi device hanno solitamente una sola funzionalità statica (single-functioned), ripetuta all'infinito, hanno stretti vincoli di prezzo, potenza e performance (tightly constrained) e devono reagire ai cambiamenti in maniera immediata senza rallentamenti (real time).

Con il passare degli anni i sistemi embedded stanno diventando sempre più potenti nonostante la loro dimensione continui a diminuire, grazie alle scoperte dell'elettronica. Inizialmente questi device venivano programmati completamente in linguaggio assembler per ottimizzare al massimo il codice e ridurre i consumi, mentre ora vengono utilizzati linguaggi di medio-basso livello e a volte si fa uso di middleware per semplificare e velocizzare lo sviluppo.

Sempre più importante è il ruolo del design, sia dal punto di vista hardware che da quello software, due aspetti che si cerca di analizzare in parallelo utilizzando tecniche di hardware-software codesign, usando come supporto linguaggi di rappresentazione che permettono di definire il sistema ad alto livello e contemporaneamente a livello hardware (VHDL, systemC).

Sistemi digitali

Sono sistemi eterogenei composti da una piattaforma HW, applicativi software, interfacce, trasduttori e attuatori. A seconda del dominio di applicazione (server PC?, telefonia?...) possono essere sistemi desktop o embedded, hanno una diversa implementazione (stile di progetto, tecnologia...) e quindi vengono programmati adeguatamente sia a livello HW (possibile riconfigurabilità) che SW (a livello applicativo ma anche a livello istruzione).

Nel caso di sistemi digitali embedded diventa necessario stabilire la reattività del sistema (deve rispondere in micro/millisecondi), l'hardware e software su cui inserire le funzionalità desiderate (FPGA, ASIC; sistemi operativi, driver), cercando di massimizzare le prestazioni riducendo il più possibile i costi e il consumo di potenza... diventa importante la progettazione.

Tramite certe tipologie di flusso di progetto è possibile analizzare il problema della realizzazione di un sistema digitale, riducendo il tempo di analisi e implementazione (spesso tramite meccanismi semi-automatici) e ottimizzando il sistema finale.

HW-SW Codesign

Un nuovo approccio si sistemi embedded (ES) è detto system-on-chip, cioè l'inserimento di vari componenti (DSP, memorie, driver... diversi processori!) su un singolo chip, dove precedentemente aveva spazio la sola CPU.
L'evoluzione del system-on-chip apre nuovi scenari che assegnano sempre più importanza ai linguaggi di specifica e alla modellizzazione di livello sistema, che permettono di progettare ad alto livello e in seguito sfruttare algoritmi automatici e componenti pre-progettati (off-the-shelf) per velocizzare e ottimizzare lo sviluppo di ES.

I problemi aperti dalla crescente complessità dei sistemi embedded sono molti, tra cui:
- come comprendere il comportamento e progettare sistemi complessi ?
- come tradurre le specifiche in implementazione?
- come controllare i vincoli real-time e la dissipazione di potenza?
- come fare testing sui sistemi, PRIMA dell'implementazione?

A queste domande cerca di rispondere l'HW-SW codesign.

L'HW-SW Codesign è una metodologia di progetto che ha come obiettivo l'ottimizzazione del flusso di progettazione, fornendo un ambiente integrato per la sintesi e validazione delle sezioni hardware e software.

Flusso di progettazione

Il flusso di progettazione (design flow) può essere parzialmente o totalmente automatizzato, usando dei tool come compilatori, programmi per ingegneria del software, e si può suddividere nella tipologia top-down (parto dal livello di astrazione più alto) o bottom-up (parto da componenti definiti a basso livello e poi adatto il resto del sistema, risalendo a componenti sempre meno dettagliati).

Il design flow più usato negli ultimi anni è appunto l'HW-SW Codesign, un flusso che enfatizza la visione unitaria tra le parti hardware e software, sfruttandone le diverse peculiarità sviluppandoli in parallelo ed evitando in questo modo errori di integrazione.

Le fasi principali del flusso di Codesign sono:
- Analisi requisiti: analizzo vincoli funzionali (descrivo output come funzione degli input, cioè definisco le funzionalità del sistema) e non funzionali (vincoli di performance, spazio, peso...).
- **Co-specifica**: modellizzazione a livello sistema con un linguaggio formale (SystemC) che permette di rappresentare sia la parte SW che quella HW, fornendo in output un eseguibile adatto al testing del sistema.
- Gestione dei task concorrenti: definisce la granularità dei processi e la loro gestione parallela.
- Trasformazioni di alto livello: elaboro ottimizzazioni di alto livello ("srotolamento" dei loop).
- Esplorazione dello spazio di progetto: valutazione delle diverse alternative di design.
- **Co-sintesi**: ottengo automaticamente l'implementazione del sistema: HW (unità dedicate), SW (processori), interfaccia HW-SW e sincronizzazione. La co-sintesi consiste in una fase di partizionamento e una di scheduling Hw-Sw:
  Partizionamento: scelgo i diversi componenti e li assegno alla sezione Sw o Hw. Sono possibili due approcci, il partizionamento *orientato all'HW* minimizza i costi creando modelli iniziali in hardware (VHDL) e in seguito portando le parti non critiche in software. L'approccio *orientato al SW* massimizza le performance con modelli iniziali software (C++), da cui poi elaboro le parti critiche in hardware.

Scheduling: identifica una relazione tra il tempo e il processo in esecuzione, scegliendo uno dei molti approcci differenti a seconda delle esigenze di progetto.

- ✗ **Co-simulazione**: controllo se il sistema rispetta le specifiche iniziali e non presenta errori. Simulo in contemporanea le sezioni HW e SW, controllandone la consistenza e validando l'integrazione tra le parti.

L'uso di questa tipologia di flusso di progetto permette di identificare tempestivamente gli errori, individuare le sezioni in cui riutilizzare codice precedentemente scritto ed effettuare test adeguati cercando di raggiungere compromessi di performance/costo.
Tutto questo aiuta a progettare meglio e più rapidamente i sistemi, fornendo enormi vantaggi ai produttori che hanno strette finestre time-to-market da rispettare.

Metriche

L'obiettivo che si vuole raggiungere progettando un sistema embedded ovviamente è quello di ottenere la funzionalità desiderata. Allo stesso tempo però si vorrebbe ottenere un sistema che ottimizzi diverse metriche di design, le più comuni sono:

- costo unitario e costi NRE (costi non ricorrenti, di progettazione);
- dimensione e peso;
- performance e consumo di potenza;
- flessibilità e mantenibilità;
- time-to-market, correttezza e sicurezza del sistema.

Spesso queste metriche contrastano tra loro e diventa necessario scegliere un compromesso, stimandole nel modo più accurato possibile.

Di fondamentale importanza sono le metriche di costo/performance/consumo di potenza e quelle di time-to-market/NRE.
Le prime stabiliscono quanto un sistema deve essere elaborato in base ai costi e al consumo, spesso è preferibile avere un sistema poco performante ma che possa funzionare per mesi con delle pile, ad esempio.
Il rapporto tra time-to-market ed NRE invece è fondamentale per evitare di entrare nel mercato in ritardo, a discapito di costi non ricorrenti più elevati per velocizzare il progetto. Se si ha conoscenza che le unità prodotte saranno poche, sarà invece preferibile avere bassi costi NRE poiché non sarà possibile coprirli nella successiva fase di vendita.

Cost Modeling

E' fondamentale progettare un sistema embedded basandosi sui vincoli imposti dalle metriche, per questo si usa una metodologia che fa ampio uso di stimatori e della modellizzazione dei costi.

I vincoli richiesti per certi applicativi sono sempre più stringenti, sia per quanto riguarda tempi di risposta che per dimensioni, peso e basso consumo di potenza.
Il flusso di prototipizzazione è spesso molto complesso e coinvolge diversi settori dell'ES, diventa sempre più importante riutilizzare dei componenti hardware/software predeterminati e testati, per ridurre il time-to-market e i costi NRE. Entrare tardi in un mercato in espansione significa perdere milioni di euro, a seconda del cost model (triangolare, a trapezio...)
Inoltre diventa sempre più importante la fase di mantenimento del prodotto e quindi è richiesta una certa flessibilità dei componenti e dell'intero progetto.

Esistono molte tecniche indipendenti usabili per stimare i costi: per analogia, top-down, bottom-up, parametrici; spesso è preferibile usarne più di una e combinare i risultati ottenuti.
Entrando nel particolare, esistono modelli parametrici software (COCOMO, permettono di stimare le metriche in modo automatico) e hardware (FPGA).

Il cost modeling è applicabile al processo di design detto RASSP (Rapid Prototyping of Application Specific Signal Processors), usando diverse metodologie...

Tecnologie

L'architettura di un sistema embedded è determinata da aspetti tecnologici come il processore (sw) o i circuiti integrati (hw) e da vincoli di progetto dovuti all'ambito applicativo e alla flessibilità richiesta.

Il processore usato in un sistema embedded può essere di diverso genere a seconda della funzionalità richiesta:

- ✔ I processori **general purpose** (GPP) sono dispositivi programmabili flessibili e poco costosi, contengono una memoria di programma e un data-path non specializzato per poter eseguire più operazioni di diverso genere (quelle fondamentali sono Load [carico da memoria ai registri], Store [salvo dai registri alla memoria] e tutte le operazioni ALU).
  E' un approccio totalmente software, non ci sono porzioni di codice implementato hardware.
  [Permette di avere basso time-to-market e grande flessibilità a discapito di un'alta potenza dissipata e tempi di esecuzione non ottimali]
  Esempi di GPP sono i microcontrollori, processori RISC, DSP e processori per multimedia.

Un'ottimizzazione di questa architettura avviene attraverso l'introduzione del concetto di pipelining, una tecnica che permette l'esecuzione contemporanea di più istruzioni secondo lo stile di una catena di montaggio. La latenza rimane invariata, ma aumenta la velocità di esecuzione di più istruzioni.
E' possibile incrementare ulteriormente le performance usando scheduling statico (VLIW) o dinamico, oppure usando processori superscalari.

- ✔ I processori **single purpose** (SPP) sono circuiti digitali creati per eseguire esattamente una tipologia di programma. Questa categoria di processori non contengono una memoria di programma, il data-path ha una logica semplice perché adatto ad eseguire solo un ridotto e statico insieme di istruzioni.
  [Sono molto veloci e consumano poca potenza, tutto eseguito via hardware, flessibilità nulla]

- ✔ I processori **application-specific** sono processori programmabili ottimizzati per una classe di applicazioni con caratteristiche comuni.
  Sono un compromesso tra i due precedenti modelli, hanno una memoria di programma, datapath ottimizzato e speciali unità funzionali. Filosofia ASIP.
  [Flessibilità, tempi di risposta e consumo di potenza buoni, ma non ottimali].

Un ASIP molto comune è il microcontrollore, usato per manipolare piccole quantità di dati nei sistemi embedded, contiene data memory e periferiche on-chip, è programmabile tramite i pin del chip.
Un altro ASIP è il Digital Signal Processor (DSP), usato nelle applicazioni che processano grandi quantità di dati, spesso in streaming, e necessitano i trasformare questi dati velocemente. I DSP hanno molte unità funzionali, ALU vettoriali, buffer...

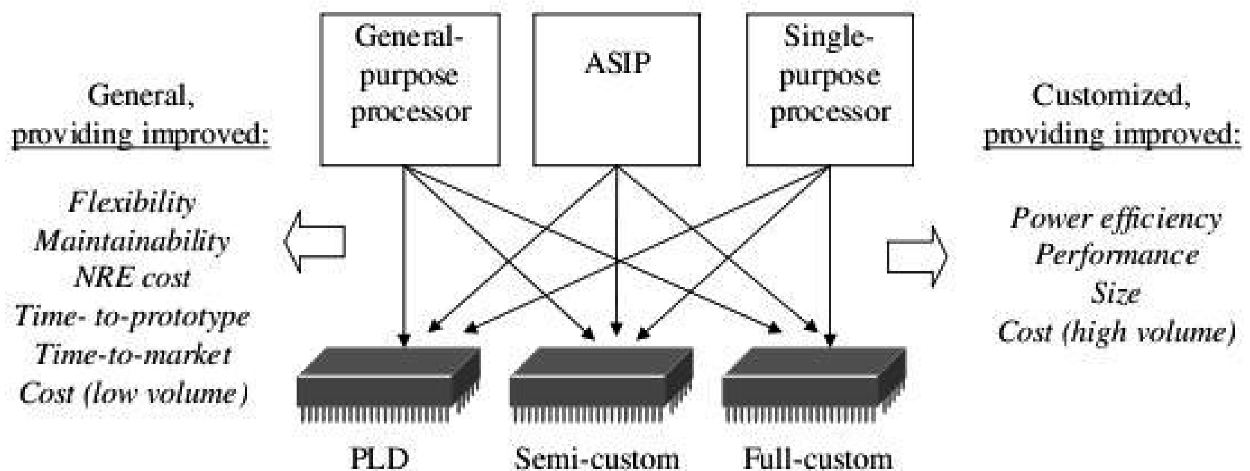
Tecnologie per circuiti integrati

Un circuito integrato è composto da più transistor, come il CMOS e il PMOS, la cui dimensione è ormai arrivata al limite fisico di 2-3 nm, per questo si stanno studiando nuove tecniche per continuare a rispettare la *legge di Moore*, che stabilisce il raddoppio del numero di transistor in un chip ogni 2 anni.

Come progettare un circuito integrato, che diventa ogni anno sempre più complesso?
Esistono tre tecnologie differenti che variano a seconda dell livello di personalizzazione dei diversi layer del circuito integrato: Full-custom, Semi-custom, Programmabel Logic Device.

- ✔ gli IC **full-custom** (VLSI) sono totalmente personalizzabili, quindi è possibile avere grandi performance, spazi ridotti e basso consumo di potenza, ma con costi NRE molto alti.
  E' necessario stabilire la posizione ed orientamento dei transistor, le loro connessioni... i circuiti così sviluppati sono tipici di una casa produttrice di dispositivi, come Intel.
- ✔ i circuiti **semi-custom** (ASIC) hanno i livelli inferiori pre-programmati, partono quindi da una struttura generale per-implementata che viene poi adattata alla particolare funzionalità digitale. Hanno costi unitari e NRE accettabili, performance e consumo di potenza buoni.
  Possono essere Gate Array (array di gate prefabbricati) o Standar Cell (librerie di celle pre-determinate, ben integrabili con sistemi full custom).
- ✔ i **circuiti programmabili** PLD sono meno costosi e semplici, mettono a disposizione componenti logici anche complessi che possono essere connessi tra loro a seconda delle esigenze di progetto.
  Prestazioni scarse, alto consumo di potenza ma hanno bassi costi NRE e time-to-market.

Un'eccessiva flessibilità può essere controproducente in termini di costo, prestazioni e consumo energetico. E' compito dell'analisi di sistema comprendere quale sia il livello di specializzazione richiesto e scegliere quindi il miglior compromesso. In seguito nella co-simulazione sarà possibile verificare se i risultati ottenuti rispecchiano le stime iniziali.



i processori si possono implementare usando le 3 tecnologie

La scelta dell'architettura è legata alle metriche di progetto ma nono solo, si basa anche sulla tipologia del sistema richiesto, che può essere "dominato dal controllo" (guidato dagli eventi) o "dominato dai dati" (hanno un flusso di dati costante, necessitano di memorie e buffer: DSP), anche se generalmente gli ES sono sistemi misti.

Microprocessori

Posso configurare un microprocessore attraversano la parallela del computer desktop, per effettuare test delle applicazioni che voglio far funzionare sul microprocessori posso caricare il codice sulla CPU oppure effettuare una simulazione, possibile usando un ISS: instruction set simulator.

Real Time OS

Introduzione

In un sistema real-time la correttezza dell'esecuzione dipende anche dal tempo in cui sono calcolati i risultati, si considera errore di sistema la violazione dei vincoli temporali.
Un sistema operativo real time (RTOS) è un elemento fondamentale del sistema RT, ma non l'unico. Esso deve assicurare che ogni processo critico avvenga entro i limiti prestabiliti, usando algoritmi adeguati alle esigenze del sistema.

I RTOS sono caratterizzati da:

- deterministic behavior: le operazioni sono eseguite in tempi predeterminati;
- responsiveness: dipende dal tempo necessario per rispondere ad un interrupt;
- user control: è più ampio che nei normali OS, l'utente può personalizzare il sistema;
- reliability: il sistema deve essere affidabile, fail-soft in caso di errori e massima stabilità.

Caratteristiche tipiche di un RTOS sono: basso costo e piccole dimensioni, latenza minima, fornisce interprocess communication, permette delay e timeout dei servizi kernel, scheduling con priorità.

L'esecuzione dei task può essere preemptive se si consente lo switch tra diversi processi, con o senza deadline... vediamone alcune implementazioni possibili.

Scheduling

Il cuore di un RTOS è lo scheduler, perchè il tempo di risposta e i vincoli di esecuzione dei singoli processi sono punti chiave di ogni sistema embedded, a causa di punti “deadline” dopo di cui l'esecuzione di un task è inutile o crea problemi (scatto di fotocamera dopo 3 secondi?).

Esistono molteplici tipologie di scheduler, ma per i sistemi RT alcuni non si rivelano adatti:

- preemptive round robin: ogni task viene messo in una coda di attesa e aspetta il suo “timeslice”, il suo quanto di tempo in cui viene eseguito.
  L'attesa di questo tempo spesso risulta inaccettabile.
- non preemptive priority: processi real time sono ad alta priorità, sono eseguiti appena il processo attualmente in esecuzione viene completato
  Se il processo in esecuzione è lento il task RT potrebbe attendere troppi secondi.
- preemption point, priority: vengono creati degli intervalli in cui un processo in esecuzione può essere interrotto se un task ad alta priorità richiede l'esecuzione.
  Buona soluzione in generale, ma non per i sistemi con vincoli strettissimi.
- preemption immediata: il servizio RT viene eseguito praticamente subito, interrompendo eventuali processi in esecuzione, rispettando quindi i vincoli dei sistemi più critici.

Uno dei fattori principali che influenza la tipologia di scheduler da scegliere è la **schedulability analysis**, che può essere statica o dinamica e si implementa con diverse classi di algoritmi:

- static table-driven: tramite analisi statica consente di determinare a run-time quando un processo deve iniziare l'esecuzione. E' un approccio non flessibile e ogni cambio di requisiti di un processo implica il calcolo di un nuovo modello di scheduling. [earliest deadline first].
- static priority-driven preempt: l'analisi statica consente di assegnare le priorità ai task in base ai vincoli temporali. [rate monotonic].
- dynamic planned-based: un task viene eseguito solo se è possibile evitare che violi i vincoli di tempo.

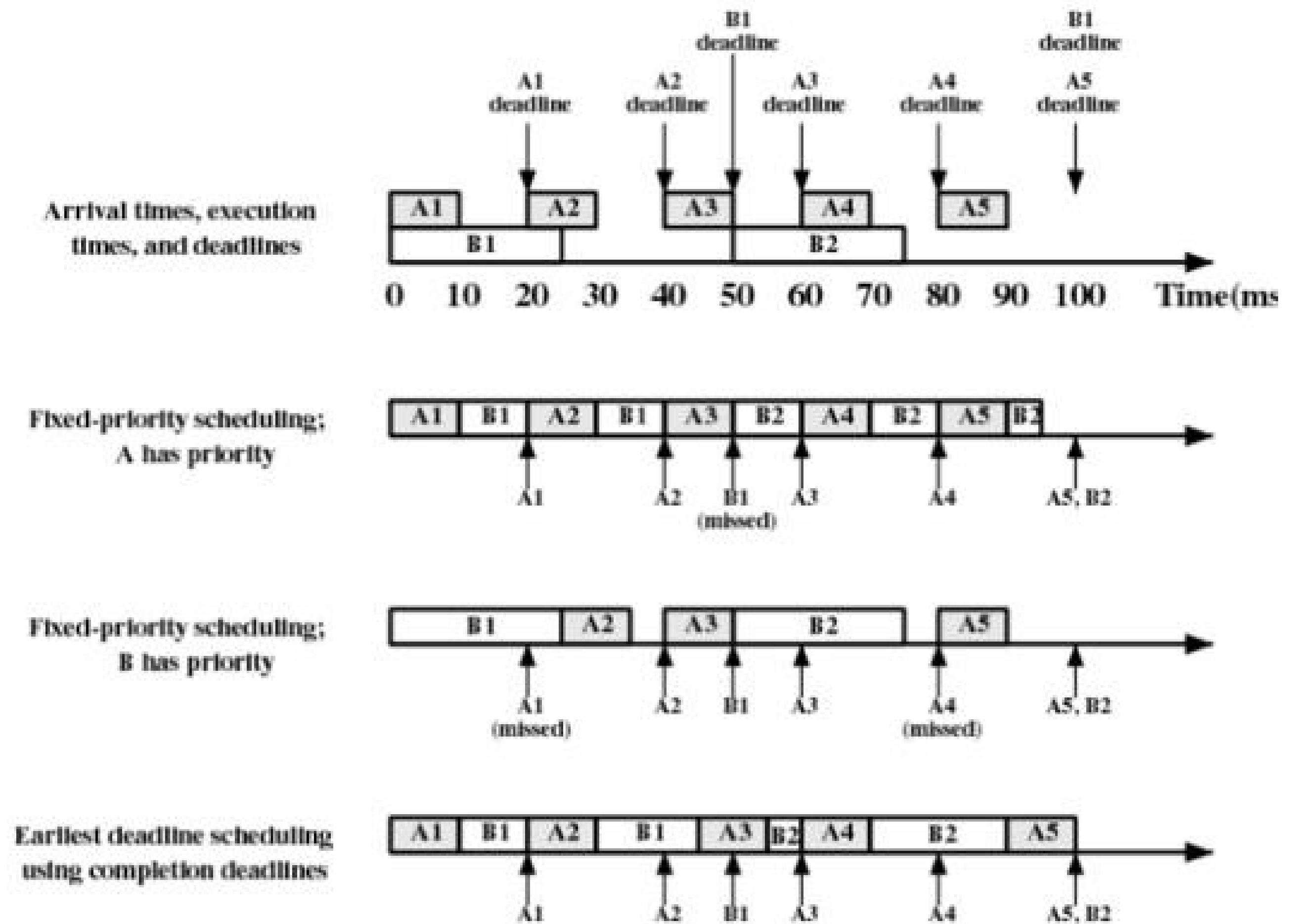
- dynamic best-effort: semplice e molto usato, all'arrivo di un task il sistema assegna una priorità basata sulle sue caratteristiche ed elimina eventuali processi che hanno violato i vincoli temporali.

- **Scheduler basati su deadline**

Gli scheduler basati su deadline usano diverse informazioni chiave: *ready time* (quando un processo è pronto per essere eseguito, conoscenza possibile se il task è periodico), *starting* o *completion deadline*, *processing time* (se non disponibile, si usano valori medi), *risorse richieste*, *prorità* ed eventuale *sotto-struttura* del processo.

Quale processo eseguire?
Nello scheduler Earliest Deadline First (EDF) viene eseguito il task con deadline più vicina, che avrà la responsabilità di lasciare l'esecuzione alla fine della sua sezione critica.
Questa tecnica permette di raggiungere obiettivi migliori che con un semplice scheduler a priorità, ma ho bisogno di maggiori informazioni e uno scheduler più complesso.
Si può raffinare l'algoritmo se è possibile conoscere quando dei processi saranno pronti all'esecuzione.

Nell'esempio seguente vediamo come sia possibile alternare 2 processi sovrapposti e con vincoli di deadline molto vicini usando l'algoritmo deadline first:



Nello scheduling Rate-monotonic (RMS) ogni task ha un periodo T e un tempo di esecuzione C; le priorità vengono assegnate in base ai periodi (maggior frequenza == maggior priorità).
Il tempo di utilizzo del processore Ci/Ti per questi processi deve essere inferiore ad un limite prestabilito che dipende dal numero n dei task: n(sqrt(2,n)-1), cioè per *n* che tende ad infinito equivale

a log(2), cioè 0.693. Questo indica che il tempo di utilizzo del processore da parte dei processi RT deve essere minore del valore limite perchè siano rispettate tutte le deadline.

Questo tempo di CPU è minore di quello raggiungibile con EDF, ma *RMS è migliore* perchè il limite superiore assicura al 100% il raggiungimento dei deadline, ma è spesso è sufficiente una percentuale minore. Inoltre con RMS è più semplice raggiungere una stabilità, viene garantito il raggiungimento delle deadline dei processi più importanti.

- **Altri scheduler**

- ✔ Cyclic executive: esiste un programma supervisore che schedula i task in base a quanto costruito nel design, eseguendo una sequenza di operazioni in modo ciclico. E' un modello semplice ma non dinamico, si adatta a sistemi che eseguono operazioni periodiche.
- ✔ Deterministic: fornisce metodi per costruire scheduler in cui la scelta dei task da schedulare è conosciuta in ogni momento. E' necessaria informazione *a priori*, sviluppa criteri per l'ottimizzazione. La creazione dello scheduler è complessa ma dinamica, difficile avere tutte le informazioni prima dell'arrivo dei task.
- ✔ Capacity-based: necessita informazioni come la quantità di tempo di esecuzione necessaria ad un task, in base alla quale viene calcolato se è possibile eseguire il processo entro i tempi previsti. Ogni task deve essere periodico, senza zone critiche. La priorità maggiore è assegnata ai task più frequenti (rate monotonic).
- ✔ Dynamic Priority: le priorità dei processi sono dinamiche, si considera lo stato attuale del sistema per decidere quale task mettere in esecuzione. Gli scheduler classici di questa tipologia sono EDF e Least-slack-time, (slack-time è il massimo tempo per cui si può far attendere un task prima che si violino i vincoli temporali).
- ✔ Imprecise results: ogni task viene diviso in optional/mandatory subtask, lo scheduler assicurerà che i sottoprocessi mandatory saranno completati entro i vincoli, mentre il resto del tempo di CPU verrà usato per l'esecuzione dei sottoprocessi opzionali.

Caratteristiche comuni nei prodotti commerciali

Per **linux** sono state aggiunte due classi per il RT, SCHED_FIFO (processi con priorità alta possono interrompere altri processi) e SCHED_RR (si aggiunge un tempo limite dopo il quale un processo deve cedere la CPU ad un task con >= priorità).

In **Unix SVR4** i processi RT sono eseguiti prima di quelli in kernel-mode, grazie alla creazione di 160 classi di priorità, in cui i task RT hanno le posizioni 100-160. Il valore della priorità è dinamico, così come il quanto di tempo in cui un processo è in esecuzione varia a seconda del livello di priorità. Sono inseriti dei punti di preemption, regioni del codice in cui tutte le strutture dati sono consistenti.

In **Windows 2000** lo scheduler è preemptive e basato sulle priorità, round robin. Esistono 2 classi di priorità, suddivise in 16 livelli in cui un processo si sposta dinamicamente.

Unix e Windows non sono sistemi adatti ai sistemi real-time, spesso si usano sistemi operativi creati su misura, come **VxWorks**. VxWorks è spesso usato in contemporanea con altri OS, gestisce le aree critiche e real time, creando un contesto per ogni task, in cui sono incluse informazioni come il timer-delay, gestori di segnali ... Usa un algoritmo preemptive con 256 priorità e un time-slice per ogni task. VxWorks gestisce interrupts hardware rapidamente, evitando context-switch grazie a delle routine ISR (interrupt service routine).
La comunicazione tra processi può avvenire in molti modi (pipe, strutture dati condivise, messaggi...), VxWorks supporta l'allocazione e rimozione di dati da diversi “memory pool” accessibili dai processi.

VxWorks fornisce tool per misurare e ottimizzare le performance di un sistema RT, contiene anche una “spy utility” contenente informazioni sull'utilizzo della CPU da parte dei diversi processi.

Windows CE è l'embedded RTOS modulare per device mobili sviluppato dalla Microsoft.
Il kernel di WinCE è completamente ricostruito, ogni applicazione consiste di un processo e più threads, che sono considerate le unità base a cui l'OS assegna il tempo di CPU.
I thread hanno una propria priorità fissa tra le 256 esistenti, sono implementati in kernel space (problemi di deadlock e race condition), esiste il time-slice che però può essere posto a 0 (il thread esegue fino al termine). Gli interrupts sono gestiti da ISRs che mettono in esecuzione il thread appropriato IST. WinCE gestisce interrupts annidati e con diversi livelli di priorità.
WinCE supporta la memoria heap e lo stack, la memoria virtuale è di 32MB per processo, dovrebbe essere usata il meno possibile perchè rallenta l'esecuzione dei task.
Anche in questo OS sono presenti tool per misurare le performance, sia per l'analisi dei tempi di interrupt (calcola le latenze), sia per lo scheduler (prendo misure in base a metriche predeterminate).

Resistenza ai guasti

La resistenza ai guasti (fault-tolerance) è la capacità di un sistema di non subire interruzioni di servizio anche in presenza di guasti. La tolleranza ai guasti è uno degli aspetti che costituiscono l'affidabilità.

I guasti (fault) possibili ad un sistema si suddividono in permanenti, intermittenti e transienti (avviene sotto particolari condizioni; possono avvenire per difetti fisici, a livello logico o a livello di sistema.

Esistono **4 meccanismi** che un'architettura di un OS affidabile deve fornire: rilevazione degli errori (si controlla se i valori in output sono accettabili), valutazione del danno (attendo e memorizzo gli eventuali probabili errori in cascata), copertura dell'errore (riporto il sistema in uno stato non errato: in avanti usando informazioni ridondanti o indietro ad un checkpoint), trattamento dell'errore (scelgo se sostituire l'elemento guasto, riconfigurarlo, ignorarlo se l'errore è ritenuto transitorio).

Hardware fault-tolerant

La resistenza ai guasti dal punto di vista hardware può essere raggiunta tramite diverse tecniche:
- Triple Modular Redundancy (TRM)

viene triplicata l'unità hardware, che vengono collegate ad un device “arbitro” che manderà in output il risultato ricevuto da almeno 2 unità replicate, in modo da resistere ad un eventuale errore.
Può essere esteso a a NRM aggiungendo unità. L'arbitro è single point of failure.
- Ridondanza dinamica

può essere cold-standby (ho più unità identiche, solo una è allacciata al sistema e tramite test periodici verifico se è guasta) o hot-standby (tutte le unità identiche sono allacciate, in caso di output diversi viene diagnosticato quale nodo sia guasto, altrimenti si prosegue. Il più comune sistema è *duplex*).
- Codici per il controllo di errori

l'individuazione di errore è molto semplice, la correzione tramite reinvio è usata su canali rumorosi, mentre la correzione totale è molto costosa e complessa, usata in sistemi particolari.
La ridondanza dipende dal modello del canale, può essere Shannon, Markov (memoria dell'ultimo stato) o altri più complessi. I codici possono essere separabili (informazione divisa dai dati di controllo) o polinomiali ciclici (i simboli dell'informazione sono coefficienti di un polinomio).

Software fault-tolerant

Tecniche software per raggiungere resistenza ai guasti:

- N-Version programming

Equivalente di NRM, costruisco N versioni indipendenti, compilate diversamente, mediante voting scelgo la soluzione “più comune” e trovo eventuali errori.
- Recovery blocks

corrisponde alla ridondanza dinamica, consta di 3 elementi software: routine primaria (esegue la parte critica), test di accettazione (controlla le uscite), routine alternativa (realizza la soluzione alternativa).

Sistemi operativi

Negli OS la tolleranza è garantita dalla presenza di 3 tabelle che permettono di classificare, localizzare e agire a determinati errori:

- valore errore: interno, CRC, timeout...
- valore locazione: rete, memoria, disco
- valore azione: aspetta e riprova, ignora, termina...

Nel caso specifico di OS/2 esiste un demone in background detto Hard Error Daemon, che monitora gli errori e si attiva di conseguenza, comunicando con il kernel, seguendo una tabella con codici dei diversi errori e relativa azione da intraprendere.

In MacOS il Gestore degli errori segue una tabella simile, ma visualizza una finestra di errore con le opzioni stop (termina il processo) e *resume* (recupera lo stato del sistema e riesegue il processo).

In Unix esiste una tabella di segnali che informano un processo di eventuale eventi avvenuti, generati dal kernel, hardware, altri processi o utenti (SIGKILL, SIGINT...).

Basi di dati

Le basi di dati si basano sulla transazione, un'unità indivisibile di istruzioni tra il comando bot (begin) e eot (end), che deve godere delle note proprietà ACID.
Il controllo dell'affidabilità si basa sulla scrittura di un log con *record di transazione* (salvo tutte le modifiche al database, passo per passo) e *record di sistema* (dump e checkpoint del sistema).

Le metodologia di recupero da un errore sono diverse: warm-restart (attuabile se si perde la memoria volatile, si risale dall'ultimo checkpoint), cold-restart (se si perde anche la memoria di massa, si parte dal dump seguendo il log).

Reti

Ad ogni livello della pila ISO/OSI è possibile attuare meccanismi di controllo e correzione di errori.
La correzione solitamente avviene tramite ARQ (automatic repeat request), di cui ne vediamo tre categorie:

- stop & wait: protocollo semplice, aspetto ACK o NAK in caso di errore. Se non si ottiene risposta, dopo un tempo “timeout” viene reinviato.
- go-back-n: risolve le lunghe attese del protocollo precedente, consentendo di ritrasmettere n pacchetti consecutivi a partire dall'ultimo consegnato correttamente.
- selective repeat: il ricevente segnala quali pacchetti sono stati ricevuti correttamente e quali no, introducendo overhead ma risparmiando tempo grazie ad una maggior quantità di info disponibili.

Esempi applicativi

– FT in **sistemi Tandem** (multiprocessore con CPU connesse in rete).

Le applicazioni usano RPC per eseguire richieste su processi server, tutti identici ma ripartiti su diverse CPU, sfruttando il multiprocessore.
Si usano messaggi e processi dedicati all'isolamento dei guasti. Non c'è un singolo punto di fallimento, in caso di guasto non ci sono blocchi del sistema, se si bloccasse un processore il carico verrebbe ripartito tra tutti gli altri, mentre la CPU con problemi si autodisabilitarebbe evitando la propagazione dell'errore.
Ogni processo è duplicato su una diversa CPU, vengono tutti sincronizzati tramite checkpoint.
Il kernel supporta processi multipli comunicanti con messaggi (memoria read-only per evitare propagazioni).
I file possono essere partizionati su più dischi e ogni partizione è duplicata su un mirror, per le transazioni si usa un device particolare che le monitora e gestisce.

– **Logged Virtual Memory** (LVM)
Fornisce un log di ogni attività di scrittura su memoria virtuale, permettendo il recupero dei dati dopo un eventuale crash.
– **Checkpoint per RTOS**
E' un protocollo per processi concorrenti, maschera gli errori in modo stabile e trasparente all'utente.
I checkpoint vengono eseguiti dopo operazioni di scrittura o dopo fork, vengono cancellati al termine del processo. Questi CK contengono tutte le pagine modificate dopo l'ultimo CK, con i suoi file aperti, stato di schedulazione, memoria condivisa ... si tracciano le dipendenze tra processi di una stessa applicazione tramite analisi di pipes, segnali, semafori...
– **FT per la consistenza di cache distribuita**
I canali insicuri dei sistemi distribuiti esigono modifiche ai classici algoritmi per il caching, usando protocolli detti leases (contratti brevi, basati su lock sincronizzati). Questo contratto dà alla cache che lo possiede diritti di scrittura su un dato per un certo periodo.
– **FT tramite comunicazioni di gruppo in LAN**
Se la FT si realizza tramite replicazione/ridondanza, è cruciale il problema della consistenza, raggiungibile tramite un sistema che permette di avere la certezza che tutti i nodi ricevano un messaggio tramite l'invio di soli 2 segnali: SendToGroup invia un messaggio M al membro *sequencer*, che inoltra in broadcast M e un numero di sequenza *s*. Se qualche nodo riceve un messaggio con *s*+1 si accorge di aver perso il messaggio *s*, che quindi può essere richiesto di nuovo.
In caso di caduta del sequencer il mittente di M si accorge e può eseguire ResetGroup che elegge un nuovo coordinatore. Adatti solo a FT di tipo hardware.
– **FT tramite gruppi di conversazione**
Il sistema consiste di processi concorrenti indipendenti, cooperativi o competitivi, che possono comunicare tra loro tramite azioni atomiche. Le *conversazioni* sono asincrone, in caso di guasto tutti i processi eseguono insieme un rollback evitando inconsistenze.
Questo tipo di FT permette non solo tolleranza a guasti HW, ma anche a guasti SW.
– **Sincronizzazione e ridondanza**
Sono due tecniche basilari per un sistema distribuito completamente FT. Chiamiamo RS (*redundancy sincronization*) la sincronizzazione di operazioni HW o SW ridondanti, utile per eliminare gli sfasamenti temporali tra le operazioni, riconoscere e recuperare i guasti.
L'implementazione di RS consiste nel controllo di processi, CPU e nodi ridondanti, controllo degli input/output, controllo di files e timer ridondanti.
Il sistema di RS è gestito da un server che stabilisce i punti di syncro, quale siano le risorse primarie, i tempi di risposta...
– **Servizi di log condiviso**
Nei sistemi distribuiti il log devono avere uno spazio di indirizzamento separato dai log locali, ma questo introduce grande overhead di comunicazione... si introduce un sistema detto Quick Silver, in cui una routine sempre attiva assegna una sequenza di interi con LSN (Log Sequence Number), i log sono gestiti localmente fino alla scrittura forzata sul Log Manager, che ne provoca il commit.
Quick Silver utilizza speciali archivi, gestiti localmente, per recuperare da eventuali guasti.
– **Memoria transazionale stabile**
L'approccio STM (stable transactional memory) sfrutta le strutture proprie delle transazioni, è composto di 3 livelli:
1°- memoria veloce e stabile per il protocollo di commit
2°- innestato su multiprocessore e architettura blandamente accoppiata (memorie private e comunicazione tramite messaggi).
3°- si innesta su un'architettura strettamente accoppiata (comunicazione tramite memoria condivisa).

Prima di accedere ad una risorsa diventa necessario passare per STM che assicura accessi e modifiche resistenti a guasti e consistenti.
– **Computazione parallela su rete di workstation**
Il problema è l'overhead di comunicazione, a cui poniamo soluzione operando su macchine virtuali poste “sopra” la macchina reale che esegue il codice scritto (tolleranza ai guasti e load balancing intrinseci!).

Lavoro su una macchina virtuale con processori sincroni mentre la macchina reale ha processori asincroni, per cui i passi paralleli sono numerati con un contatore logico che permette di definire quale CPU esegue un thread, evitando problemi e tollerando guasti su qualunque tipo di workstat.
Il processo di compilazione consiste in un prefase di conversione del codice e una successiva creazione di una tabella detta “progress table” (dove salvo thread e segnalo se è eseguito o terminato).

- **Dischi RAID**

Sono Redundant array of independent disk che operano in parallelo. Sono di diverso tipo: RAID 0-5, comunque i dati vengono ripartiti su più dischi e la ridondanza introdotta permette di effettuare ripristino in caso di guasto:

RAID 0: niente ridondanza, tutti i dati sono distribuiti sull'array di dischi tramite striping (distribuzione uniforme dei dati sui dischi, per load balancing)
RAID 1: eseguo mirroring dei dischi, raddoppiano costi ma anche la sicurezza (e ho *read* veloci).
RAID 2: array ad accesso parallelo e simultaneo, rilevazione e correzione errori tramite codice Hamming, bastano logN dischi di ridondanza.
RAID 3: un solo disco di ridondanza, di sola parità. La simulteneità dà tempi lenti di risposta.
RAID 4: accesso indipendente ai dischi, posso gestire in parallelo più richieste. Introduco overhead per le scritture, e il disco di parità è collo di bottiglia.
RAID 5: elimino il problema del bottleneck distribuendo le stripe di parità con algoritmo RR.
RAID 6: usa 2 stripe di parità che possono correggere due eventuali guasti contemporanei.
RAID 7: fino a 3 stripe di parità, si usa un RTOS dedicato che genera la parità e ricostruisce le informazioni, rendendo altamente indipendenti tutti i dischi (costo elevato!!).