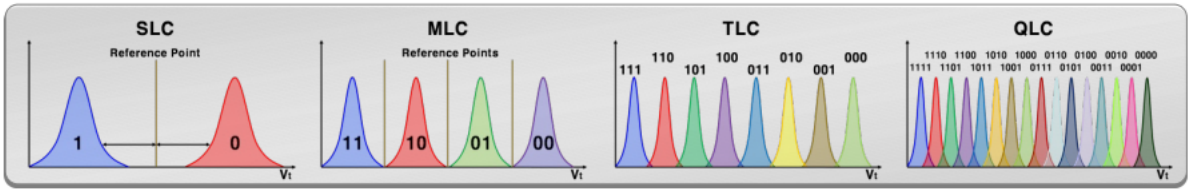
SSD (Solid State Disk)

I dischi meccanici avevano un problema fisico, che limitava la velocità con cui possiamo effettuare I/O, così negli anni ‘80 Toshiba ha ideato le memorie flash, che memorizza in maniera persistente i dati come la RAM ovvero con transistor. Le memorie NAND vanno bene per accessi sequenziali ed è per questo che vengono usate dentro gli SSD mentre i NOR dentro la RAM perchè vanno meglio per gli accessi casuali.

Salvataggio di un singolo bit

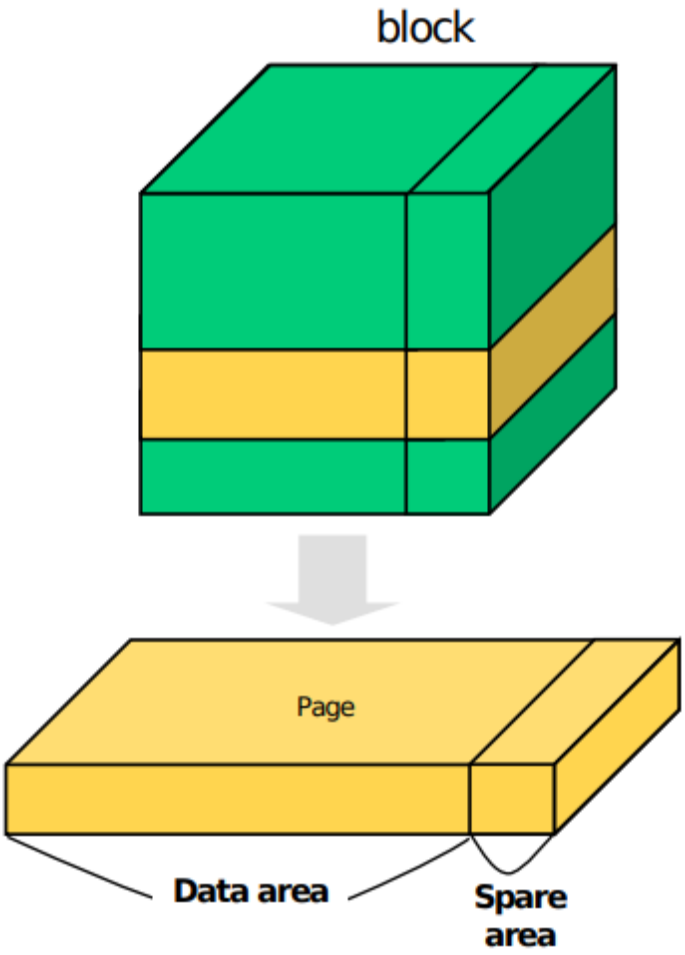
L’idea alla base è che in questi chip **ciascun transistor può salvare 1 o più bit** dentro di esso, si mantiene un certo voltaggio che mappa un valore binario. Con diversi livelli:

- **SLC** → single level cell ovvero 1 bit per transistor
- **MLC** → multi level cell ovvero 2 bit per transistor
- **TLC** → triple level cell ovvero 3 bit per transistor
- **QLC** → quadruple level cell ovvero 4 bit per transistor

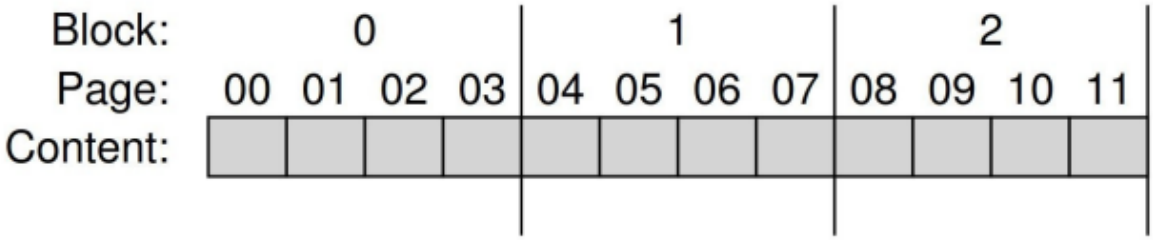


Ogni volta che vogliamo salvare più bit su un transistor, aumenta la precisione con cui dobbiamo leggere il voltaggio, per un TLC per esempio ci sono 8 livelli di voltaggio e per il QLC 16, per questo non andiamo oltre perchè leggere più livelli di voltaggio risulta pressoché impossibile. SLC è già sufficiente.

I chip flash sono organizzati a **banchi**, ai banchi possiamo accedere in due modi, **blocchi** o **pagine**, il primo prende tutto il blocco con dimensione 128-256Kib, le pagine invece 4Kib.



Così facendo possiamo indirizzare o un blocco o una pagina per lettura o scrittura. Ad esempio un banco ha tre blocchi e ogni blocco ha quattro pagine:



Abbiamo 3 operazioni principali:

- **read**(pagina): usata per leggere il contenuto di una pagina
- **erase**(blocco): usata per cancellare un blocco, prima di scrivere su un blocco dobbiamo eliminare completamente il suo contenuto, quindi mettiamo a 1 tutti i bit del blocco.
- **program**(pagina): usata per scrivere una pagina

Ogni pagina ha uno stato associato:

- **INVALID**: stato iniziale
- **ERASED**: il blocco è stato cancellato
- **VALID**: i dati sono stati impostati

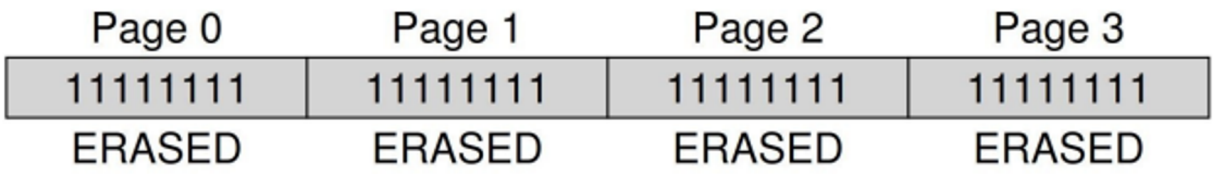
Esempio di operazioni su un blocco di 4 pagine, “iiii”:

- Erase() → **EEEE**, lo stato delle pagine del blocco
- Program(0) → **VEEE**, la prima pagina viene impostata come valida
- Program(0) → **errore**, non possiamo riscrivere senza aver eliminato
- Program(1) → **VVEE**, ora anche la seconda pagina è valida
- Erase() → **EEEE**, così facendo eliminiamo il blocco

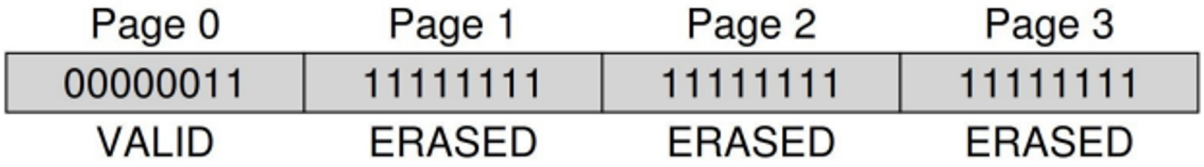
Esempio pratico, con pagine già programmate in precedenza:

Page 0	Page 1	Page 2	Page 3
00011000	11001110	00000001	00111111
VALID	VALID	VALID	VALID

A questo punto vogliamo scrivere la pagina 0, prima di farlo dobbiamo eliminare tutto il contenuto delle pagine:



E dopo aver eliminato il contenuto possiamo scrivere il dato 00000011“ nella pagina 0:



Come possiamo fare quindi per mantenere le informazioni delle altre pagine? Non vogliamo perdere dati quindi dobbiamo svolgere un salvataggio momentaneo.

Le read costano poco come operazioni, con il segnale elettrico basta andare a quel transistor, per le write invece abbiamo un tempo assai più lungo perchè dobbiamo oltre a fare la program, dobbiamo anche fare la erase.

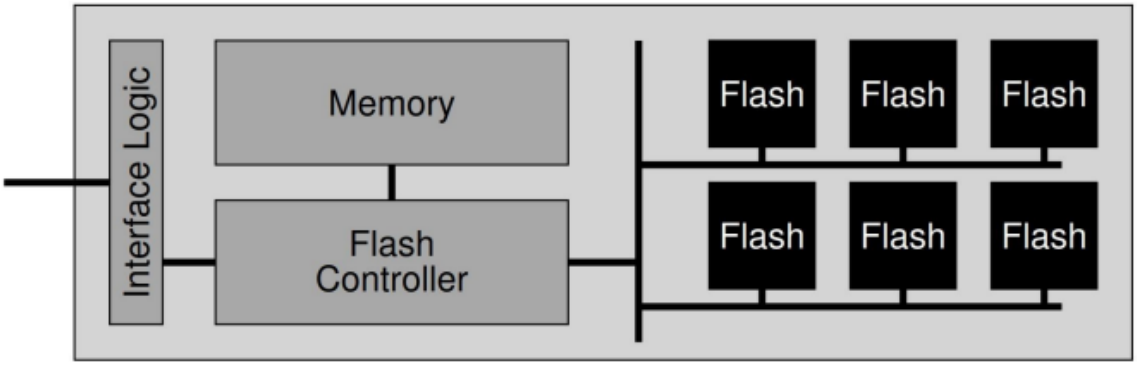
Lungo il tempo, quando un blocco viene eliminato, il transistor mantiene un po’ di carica e quindi il blocco dopo molto tempo viene reso inutilizzabile il blocco in memoria.

Il ciclo di vita di una memoria SDD viene calcolato su cicli Program Erase, e sono i seguenti dati:

- SLC 100.000 P/E cicli
- MLC 10.000 P/E
- TLC 3.000 P/E

Questo spiega perchè SLC dura di più, sperimentalmente però si è dimostrato che la durata è assai più elevata.

Il problema della **disturbance** quando leggiamo un blocco è possibile che bit di blocchi vicini vengano invertiti, abbiamo quindi una **read/write disturbance.**



Una funziona principale del controller logico (Interface Logic) è il **Flash Translation Layer (FTL ovvero un programma dentro il controller sulla motherboard)**, quando arrivano indirizzi di settore, questa interfaccia mappa l'indirizzo sulla pagina ancora logico (l'Interface Controller). Il Flash Controller prende la pagina giusta eseguendo una traduzione dell'operazione in arrivo in operazioni P/E.

Questo layer ha due obiettivi: **parallelismo** e ridurre una **amplificazione delle scritture** ovvero la quantità di byte scritti e quanti byte sono mossi dentro l'SDD (byte mossi / byte scritti = amplificazione della scrittura).

Dobbiamo però ridurre i cicli P/E per evitare il problema della condensazione di energia, FLT deve quindi poter distribuire le write in modo uniforme così da ridurre l'usura (**wear leveling**).

Il programma FLT minimizza la **program disturbance** delle write, partendo da pagine più basse a pagine più alte così da ridurre i flip dei bit.

Mappatura Diretta FTL

Una lettura di una pagina logica N è direttamente mappata su una pagina fisica N, quando però scriviamo su una pagina logica N, dobbiamo fare la erase prima della pagina fisica N. Siccome non vogliamo perdere il contenuto delle altre pagine, prima leggiamo il contenuto, poi erase del blocco e poi facciamo il program di pagine vecchie e pagina nuova, ma il tutto ha un costo. Avremo un problema di prestazioni e quindi FLT con mappatura diretta è una cattiva idea

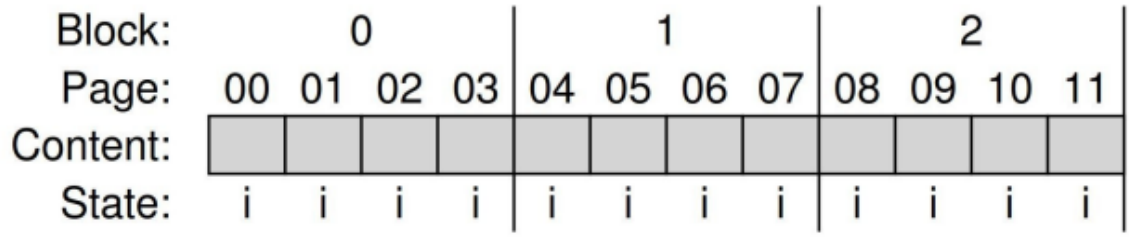
Log-structured FLT

Una pagina logica N, FTL esegue una operazione di write nella prossima pagina libera nel blocco che sta scrivendo, quindi prima di andare in un altro blocco dobbiamo scrivere tutto un blocco. Dobbiamo mantenere una tabella per mappare queste associazioni

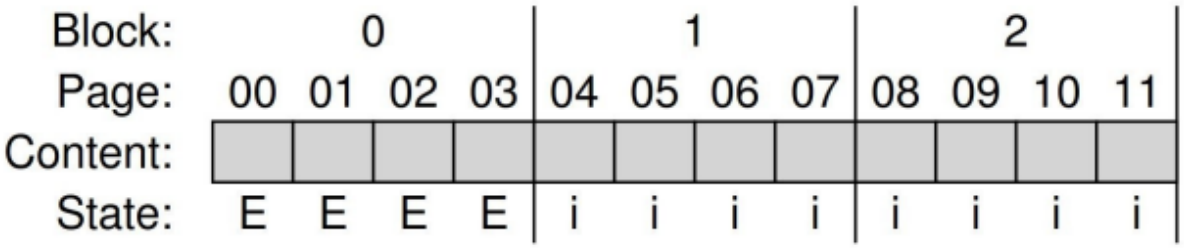
Esempio:

- Il SO fa operazioni di 4KB
- Il SSD contiene blocchi da 16KB divisi in blocchi da 4KB
- Eseguiamo le seguenti operazioni:
  - write(100)
  - write(101)
  - write(2000)
  - write(2001)
- Dobbiamo scegliere in quali pagine metterle:

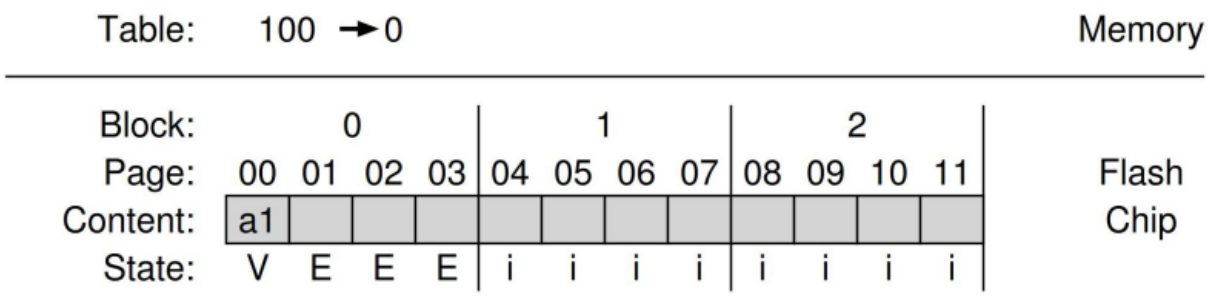
Lo stato iniziale del SSD, dobbiamo prima di tutto eliminare il contenuto, lo stato dei bit deve essere E invece che i:



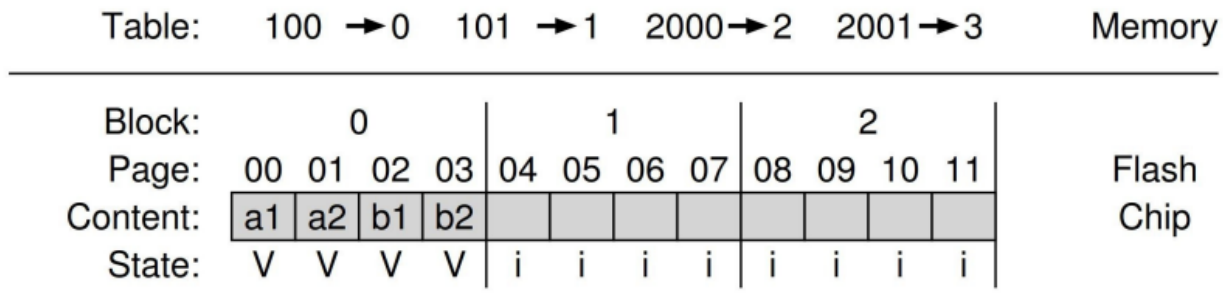
Dopo la erase abbiamo questa situazione:



A questo punto abbiamo 4 pagine libere, leggiamo nel log qual è la prima operazione da eseguire:

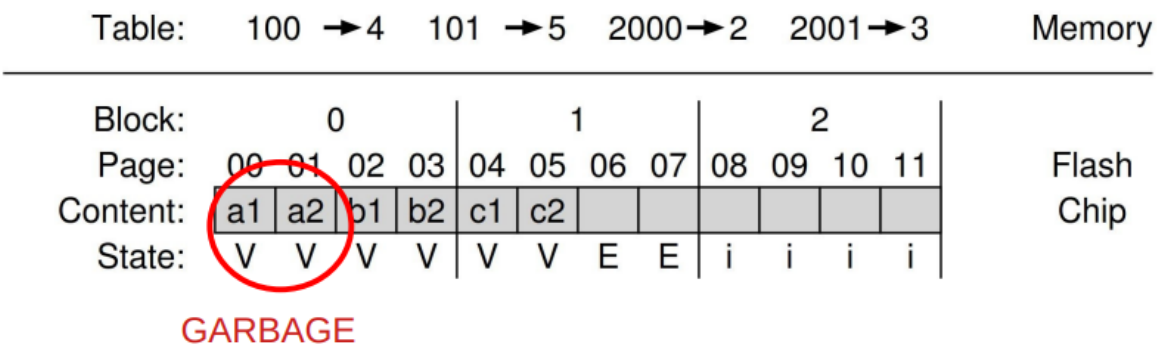


Cancelliamo dal log quindi l'operazione e nella tabella lasciamo una nuova entry della mappatura quindi 100 → 0 ovvero la pagina fisica 100 è nel blocco 0, pagina 0.

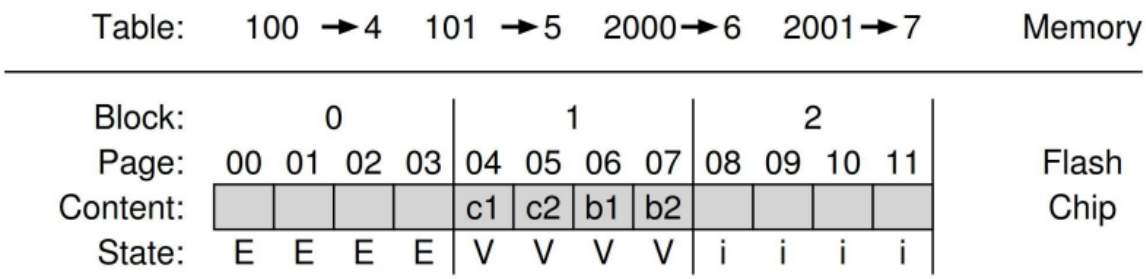


Dobbiamo eseguire questa operazione fin tanto che il blocco non è pieno, mantenendo nella tabella di mappatura **posizioneFisica → posizioneLogica**.

Così facendo è tutto concentrato in un blocco, possiamo fare wear-leveling ma abbiamo anche dei problemi ovvero dobbiamo mantenere una tabella in memoria per avere la mappatura. Il problema è che si crea della **garbage** all'interno dei blocchi, ad esempio se i blocchi 0 e 1 hanno nuovi dati saranno scritti nelle prime due posizioni del blocco 1, e quelle nel blocco 0 diventano **garbage**.



A questo punto possiamo fare in modo di eliminare il contenuto, questa azione è possibile grazie alla tabella che salva la mappatura delle pagine logiche in pagine fisiche. Tutte le celle vecchie sono etichettate come **garbage** e viene chiamato questo processo **garbage collection**.



@todo finire appunti da 385 a 400

Consumo energetico

I sistemi con computer e monitor consumano in media 1200 Watt, come possiamo ridurre per avere un impatto più sano sull'ambiente e aumentare la durabilità della batteria?

- **Livello hardware**: costruire device green che consentono all'hardware di spegnersi quando non in uso.
- **Livello software**: si dota il SO di politiche che spengono le varie periferiche quando non in uso.

Ci sono diversi stati energetici:

- On: acceso, funzionamento a pieno regime

- **Sleeping**: l'uso del device è sospeso per breve tempo
- **Hibernating**: l'uso del device è sospeso per molto tempo, con l'ibernazione si salva più energia ma ci mette di più la ripartenza, con la sleeping invece si consuma di più ma ci mette meno a riprendere
- **Off**: il dispositivo non consuma energia

Il SO deve gestire tre decisioni:

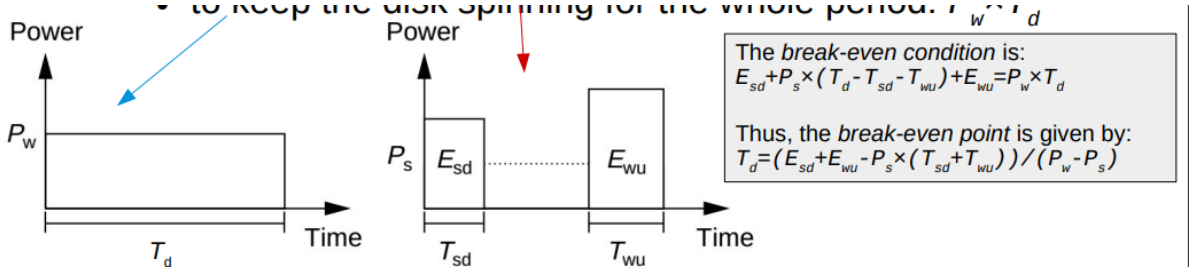
- Decidere **quale** device mettere in quello stato
- Decidere **quando** mettere il device in quello stato
- Decidere **come** mettere il device in quello stato

Un esempio è come scegliere il dispositivo da spegnere, se il SO sceglie un dispositivo che sta per essere usato allora abbiamo un rallentamento al contrario se aspetta troppo a decidere avrà perso energia, ci vuole una stima quindi.

Gli algoritmi devono prendere una decisione su diversi input che sono: **come**, **quando** e **chi**.

Esempi di dispositivi

- Il **monitor**: quando non c'è attività per alcuni minuti possiamo decidere se spegnere o meno il monitor, spegnere il monitor lo mette in sleep così da non perdere il valore della vram.
- **L'hard-disk**: usato frequentemente, quindi quando non sta più girando allora significa che allora possiamo metterlo in idle o persino spegnere il dispositivo. Attenzione però che la ripartenza dell'hard disk consuma molta energia e rallentarlo / ripartire con molta frequenza potrebbe peggiorare il suo lifetime.
    - **Punto di pareggio:** ovvero quando accendere un disco è uguale all'energia che serve per spegnerlo (?)



- **Esd** = Energy speed down → quanto serve per rallentare
- **Ps** = Power sleeping → quanta energia consumo in sleeping
- **Tsd** = Time speed down
- **Twu** = Time wake up

Se quindi tenerlo acceso per Pw (Power wake) il punto di pareggio è:

Thus, the *break-even point* is given by:
$T_d=(E_{sd}+E_{wu}-P_s\times(T_{sd}+T_{wu}))/(P_w-P_s)$

@todo riscrivere la formula

Esempio con numeri

- EXERCISE: consider an HDD with two energy states (AWAKE and ASLEEP) such that:
  - P_W = 94.3 W
  - P_s = 59.8 W
  - E_{SD} = 726.1 J, T_{SD} = 7.7 s.
  - E_{WU} = 2885.6 J, T_{WU} = 30.6 s.
  - T_U = 42.8 s. (for the next T_U s. the HDD will not be used)
- Is it better to put the HDD in the ASLEEP state, to keep it in AWAKE, or is it the same?
- What is a value T_U' that inverts the previous answer?

- Compute the break-even point
  - $T_d=(E_{sd}+E_{wu}-P_s\times(T_{sd}+T_{wu}))/(P_w-P_s)$ =
  - $(726.1+2885.6-59.8*(7.7+30.6))/(94.3-59.8)$ =
  - $(3611.7-59.8*38.3)/34.5$ =
  - $(3611.7-2290.34)/34.5 = 1321.36/34.5$ =
  - $38.30002898...$
- We have thus that $T_U < T_d$ so it is better to put the HDD in the ASLEEP state
- T_U' = any value smaller than T_d, e.g. T_U'=34.5s inverts the previous answer, i.e. it is better to keep the HDD in the AWAKE state

Tu è maggiore il prof ha sbagliato nella slide, va comunque il ASLEEP è solo sbagliato il verso. Con qualunque valore minore di Td il nuovo Tu_1 andrà bene per mettere il dispositivo in AWAKE.

La stessa idea può essere applicata al processore

- CPU power consumption *P* can be modeled as:
  $P = P_{dyn} + P_{static}$
- On many CPUs, P_{dyn} is proportional to the square of the CPU voltage *V* and to the clock frequency *f*
  $P = k\times V^2\times f + P_{static}$
  - *k*: constant depending by low-level system design factors
- Idea: reduce *V* and/or *f* to reduce *P*
  → *Dynamic Voltage and Frequency Scaling* (*DVFS*)
  - Reducing the voltage will also reduce clock frequency (approximately linearly)

Voltaggio e freqeunza sono linearmenet dipendenti!!!

- EXERCISE: consider a system where:
  - there is only one process *P* running
  - *P* requests the CPU every T_R=6.7s
  - the CPU takes T_P=2.226s to process the request
  - P_{static} = 0 and P_{dyn} = 51.1 W
- If the max voltage *V* is cut to *V/n*, what is the optimal value of *n* that results in the lowest energy consumption without causing queueing for the requests?

- $E_{no\ cut} = n_{req} *(P_{dyn}*T_P+P_{static}*T_R)=n_{req}*P_{dyn}*T_P$ = 10*51.1*2.226 = 1137.486 W
- $E_{cut} = n_{req}*((P_{dyn}/n^2)*T_P*n)$ = 10*(51.1/9)*2.226*3 = 10*5.67777*6.678 = 379.162 W
- $D_{E\%} = ((E_{no\ cut} - E_{cut})/E_{no\ cut})*100\%$ = (758.323/1137.486)*100% = (0.666666)*100% = 66.6666%