Risultati dei test sul filesystem

Abbiamo effettuato svariate ricerche sul web per individuare un test che potesse essere adatto alla valutazione delle prestazioni del nostro filesystem.
Alla fine la scelta è ricaduta sulla nota suite di benchmark **Bonnie++** (http://www.coker.com.au/bonnie++/).
Questa si focalizza sulle velocità di lettura e scrittura dei dati e quindi è apparsa subito come perfetta per le nostre esigenze, anche grazie alla sua leggerezza e semplicità di utilizzo.

Vediamo in dettaglio i test che sono stati affrontati e discutiamone le prestazioni:

1. **Writing a byte at a time**
   Questo test scrive su un file utilizzando la funzione *putc*. Qui viene sollecitata principalmente la CPU, perché i dati vengono assorbiti dalla cache
2. **Writing intelligently**
   Questa volta il file viene scritto utilizzando una *write*. Il carico richiesto alla CPU consiste solamente nell'allocazione del file da parte del sistema operativo
3. **Rewriting**
   Ogni porzione del file pari alla dimensione del buffer viene letta con una *read*, modificata e poi riscritta con una *write* utilizzando anche una *lseek*. Poiché non viene fatta alcuna allocazione dello spazio e l'I/O è localizzato, questo test verifica il corretto funzionamento della cache e la velocità del trasferimento dati
4. **Reading a byte at a time**
   Il file viene letto utilizzando *getc*. Questo mette sotto sforzo solo *stdio* e l'input sequenziale
5. **Reading intelligently**
   In questo caso vengono valutate le prestazioni del filesystem per la lettura di file usando la funzione *read*
6. **Random seeks**
   Questo test avvia numerosi processi che effettuano diverse *lseek* in posizioni casuali del file per poi leggerle con la funzione *read*. Nel 10% dei casi la porzione di file viene modificata e riscritta con una *write*. L'idea dietro l'uso di questi processi, è quella di essere sicuri di avere sempre una *seek* in coda
7. **Create files in sequential order**
   Questo test crea una grande quantità di file utilizzando delle *create*. Questi hanno nomi che cominciano con sette numeri seguiti da un numero casuale (da zero a dodici) di caratteri alfanumerici.
8. **Stat files in sequential order**
   Richiama *stat* su ogni file
9. **Delete files in sequential order**
   I file precedentemente creati vengono eliminati con l'operazione *unlink*
10. **Create files in random order**
    Il processo è simile a quello della creazione in ordine sequenziale, solo che i nomi dei file presentano prima i caratteri casuali e poi quelli fissi
11. **Stat files in random order**
    Si richiama *stat* sui file appena creati
12. **Delete files in random order**
    Si procede all'eliminazione dei file in ordine casuale

In particolare, i primi sei vogliono emulare tutte quelle attività che risultano essere il collo di bottiglia per applicazioni I/O intensive. Gli ultimi sei, invece, coinvolgono le funzioni *create/stat/unlink* per simulare operazioni che rappresentano il collo di bottiglia per macchine in cui si hanno migliaia di file in una cartella.

Vediamo i risultati:

Version 1.97	**Sequential Output**						**Sequential Input**				**Random Seeks**			**Sequential Create**						**Random Create**					
size	Per Char		Block		Rewrite		Per Char		Block				Num Files	Create		Read		Delete		Create		Read		Delete	
	K/sec	% CPU	K/sec	% CPU	K/sec	% CPU	K/sec	% CPU	K/sec	% CPU	/sec	% CPU		/sec	% CPU	/sec	% CPU	/sec	% CPU	/sec	% CPU	/sec	% CPU	/sec	% CPU
4M	9	26	+++	++	+++	++	1092	99	++	++	13694	80	5	32	0	+++	++	35	0	31	0	+++	++	35	0
Latency	937ms		2397us		8074us		8058us		2506us		259ms			103ms		22825us		118ms		82325us		4032us		60821us	

Per valutarli faremo riferimento anche al seguente test sul filesystem ext4:

Version 1.97	**Sequential Output**						**Sequential Input**				**Random Seeks**			**Sequential Create**						**Random Create**					
size	Per Char		Block		Rewrite		Per Char		Block				Num Files	Create		Read		Delete		Create		Read		Delete	
	K/sec	% CPU	K/sec	% CPU	K/sec	% CPU	K/sec	% CPU	K/sec	% CPU	/sec	% CPU		/sec	% CPU	/sec	% CPU	/sec	% CPU	/sec	% CPU	/sec	% CPU	/sec	% CPU
4M	303	98	+++	++	+++	++	1031	99	++	++	++++	++	5	+++	++	+++	++	++	++	++	++	+++	++	++	++
Latency	31805us		66us		72us		8020us		17us		20038us			424us		1133us		1161us		455us		22us		422us	

La riga più interessante è quella relativa alla latenza. In effetti possiamo notare come vi siano enormi differenze tra i due filesystem (anche di diversi ordini di grandezza).
Infatti dobbiamo ricordare che FUSE si interpone tra l'utente ed il kernel traducendo le istruzioni del primo in equivalenti istruzioni privilegiate. Tale processo implica quindi un consistente overhead con conseguente perdita di efficienza. Inoltre parte della perdita di performance è dovuta anche alla struttura stessa di HashFS. Si può poi notare come questa sia maggiore per le scritture; ciò avviene perché ogni volta che viene modificato un file si procede al ricalcolo dell'hash. Anche la fase di creazione dei file risente di tale problema, inoltre durante l'esecuzione dei test si sono notati rallentamenti progressivi man mano che si procedeva. Questo perché all'aumentare del numero di file all'interno della directory, aumentava anche il tempo di calcolo per il suo hash, in quanto è necessario analizzare il suo contenuto.

Per quanto detto fino ad adesso, appare chiaro come sia estremamente complicato aumentare le prestazioni mantenendo la medesima tecnologia. Probabilmente l'unica strada percorribile consiste nell'implementazione di questo filesystem a livello del kernel linux, sebbene ciò comporti maggiori oneri per lo sviluppo.