Test Report for Real Time File System

Guo Jiahua PB10000614

Li Bojie PB10000603

Hu Jianwei PB10011033

2012-06-17

Contents

1 Overview

1.1 Test Cases

There are three binary variables among our benchmarks:

- Interval between two requests (*sparse* or *dense*). Sparse means that real-time thread "sleep" some time between completion of a request and submission of next request. Dense means that a new request is submitted as soon as the previous request completes.
- Block position to be accessed (*random* or *sequential*). In random cases, positions of blocks to be read are generated randomly, hence data locality is weak. In sequential cases, a large file (say, 1GB) is read sequentially, hence readahead mechanism will make sense.
- Other non-real-time tasks (*with* or *without*). This measures the extent to which the file system is able to distinguish real-time and non-real-time tasks. Our non-real-time tasks are 16 user-mode threads reading a large file (The file is not accessed by real-time task).

For each case where task interval is sparse or dense, and block position is random or sequential, there will be four test results where the code is original or modified, and with or without non-real-time tasks.

1.2 Goal

In theory, there should not be much difference for real-time metrics no matter how many non-real-time tasks exist. Hence there should be much difference between original and modified code for cases with non-real-time tasks, showing the improvement we have done for real-time disk system.

The two propositions we are going to validate are:

1. There is much difference between original and modified code, both with non-real-time tasks.
2. There is no significant difference between cases with and without non-real-time tasks, both with modified code.

Hypothesis testing should be used (by rejecting the corresponding null hypotheses) under a certain significance level. But we do not have enough time to do these calculation, and the statistics and graphs themselves are enough convincing.

1.3 Metrics

The following metrics are reported for each benchmark:

1. Average (Avg.) response time of requests
2. RootMean Square (RMS) of response time of requests (Requests with high response time are highlighted in this metric, since real-time systems are sensitive of slow responses.)

3. Standard Deviation (σ) of response time of requests
4. Response time distribution graph of requests (Log-scale is used for a clearer view; In most graphs requests are ranked by response time to form an increasing curve.)

1.4 Test Environment

Hardware: (Real Machine)

- Pentium III CopperMine (998.937MHz, i686)
- 512MB DDR2 Memory, 515524KB Useable
- 80GB Hard Disk, 19GB ext2 partition for testing

Software:

- slackware 10.0 (linux 2.4.29 kernel)
- rtlinux-3.2-wr

Each time before a test case runs, the machine is rebooted to clear memory cache.

2 Data & Analysis

2.1 Random Dense

2.1.1 Statistics

- All statistics are rounded to microseconds.
- *ID* is the identifier of a test case (see 4.1.1).
- *Code* is *RT* (Real-time) or *Orig* (Original).
- *Non-RT* is the number of non-real-time tasks.
- The meanings of statistics have been shown in 1.3.

ID	Code	Non-RT	Avg.	σ	RMS
t1	Orig	–	7.72	2.83	8.22
b2	RT	–	7.74	2.80	8.23
t3	Orig	1	15.32	4.47	15.96
b3	RT	1	10.65	4.91	11.73
t6	Orig	4	34.47	15.15	37.66
b5	RT	4	8.63	4.48	9.72

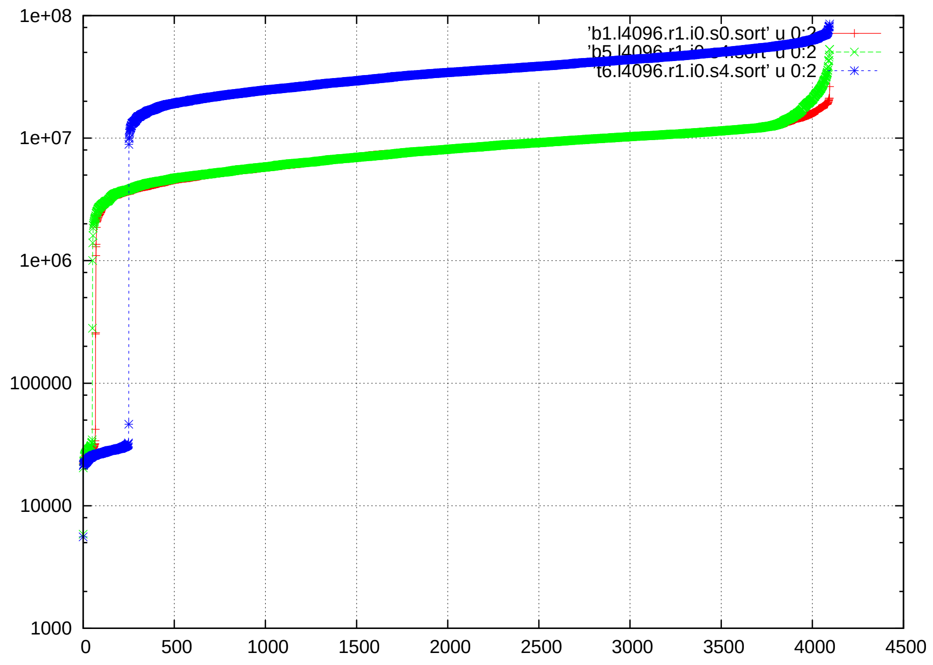
2.1.2 Graph comparing RT with original

The response time distribution graph is generated by sorting 4096 response times in increasing order and drawing a linespoints curve with GNUPLOT. Note that the response times are shown in log-scale.

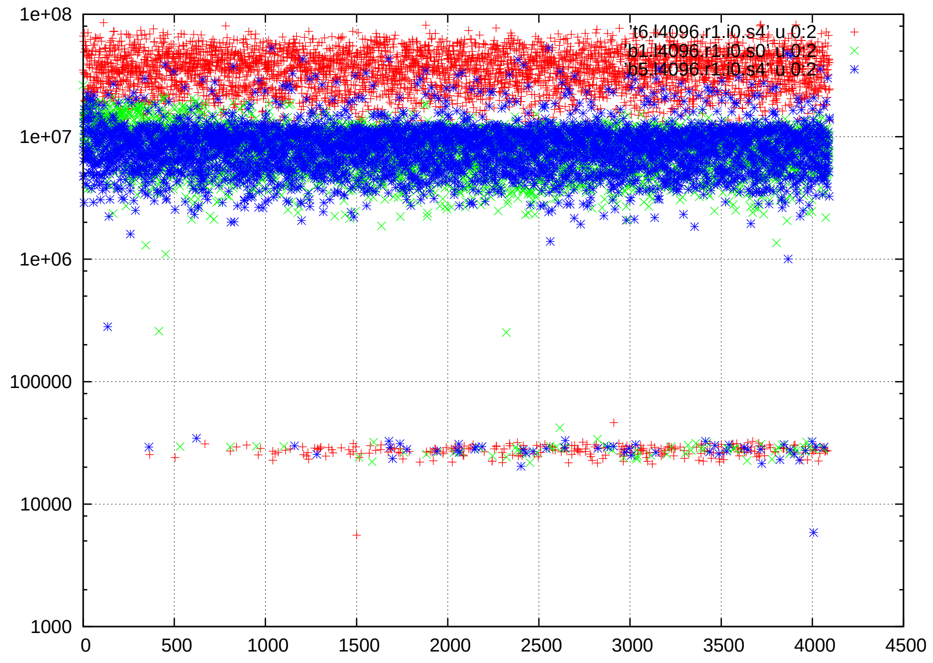
In the following graph three cases

- Red: RT code, no non-RT tasks
- Blue: Original code, 4 non-RT tasks
- Green: RT code, 4 non-RT tasks

are shown for comparison.



Here goes the distribution of unsorted response time (meanings of colors are different with the sorted version):



It should be clear that the following two statements hold:

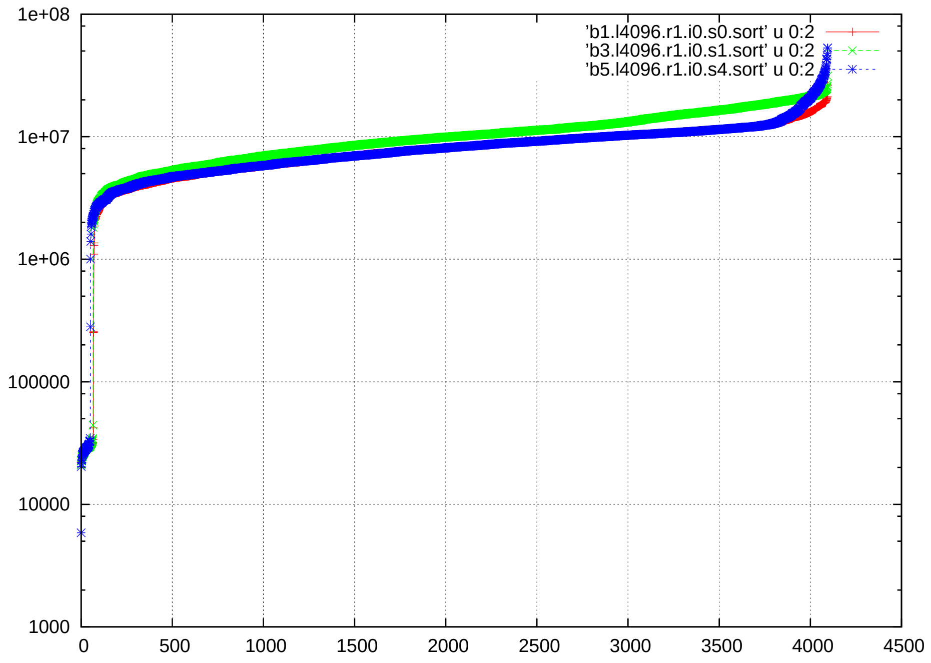
- There is much difference between original and modified code, both with non-real-time tasks.
- There is no significant difference between cases with and without non-real-

time tasks, both with modified code.

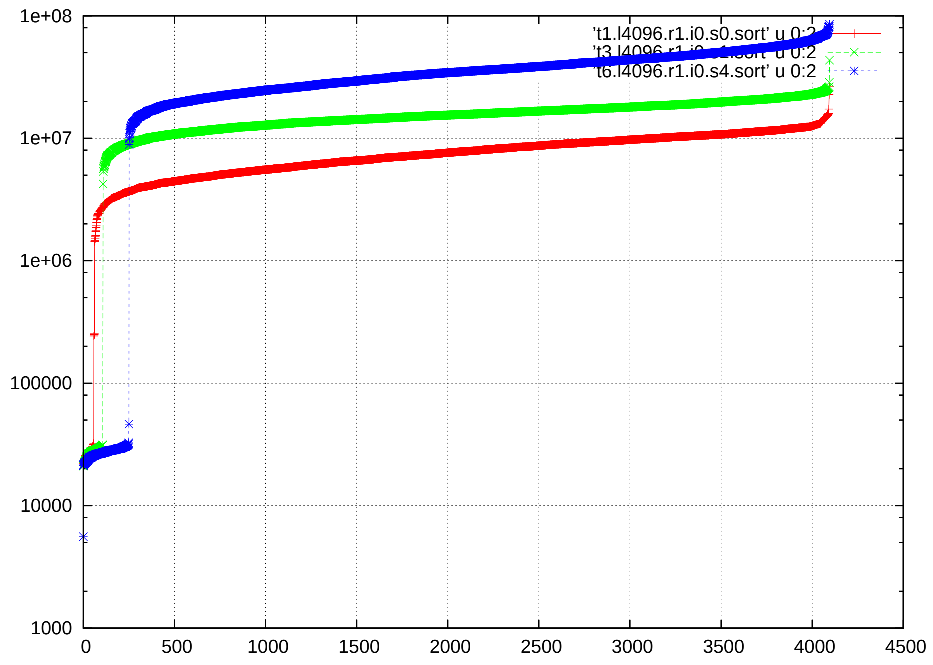
2.1.3 Graph with various number of non-RT tasks

The following graph shows how the real-time code performs across different numbers of non-real-time tasks:

- Red: no non-RT tasks
- Green: 1 non-RT task
- Blue: 4 non-RT tasks



As comparison, the following graph shows how the original code performs across different numbers of non-real-time tasks:



It is obvious that the number of non-real-time tasks does not influence real-time performance significantly in our real-time disk system, while the original system is sensitive to non-real-time tasks.

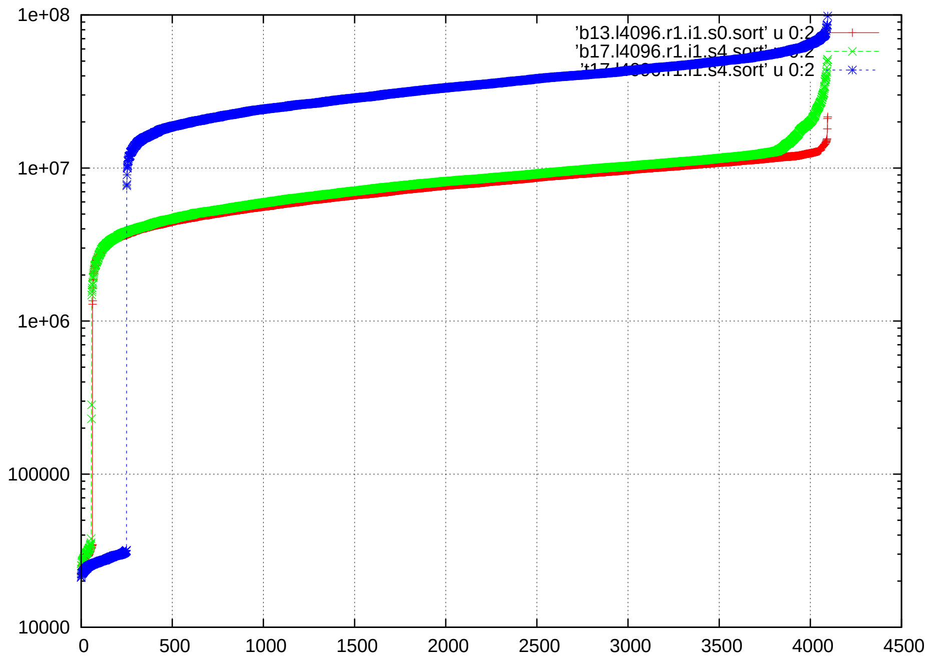
2.2 Random Sparse

2.2.1 Statistics

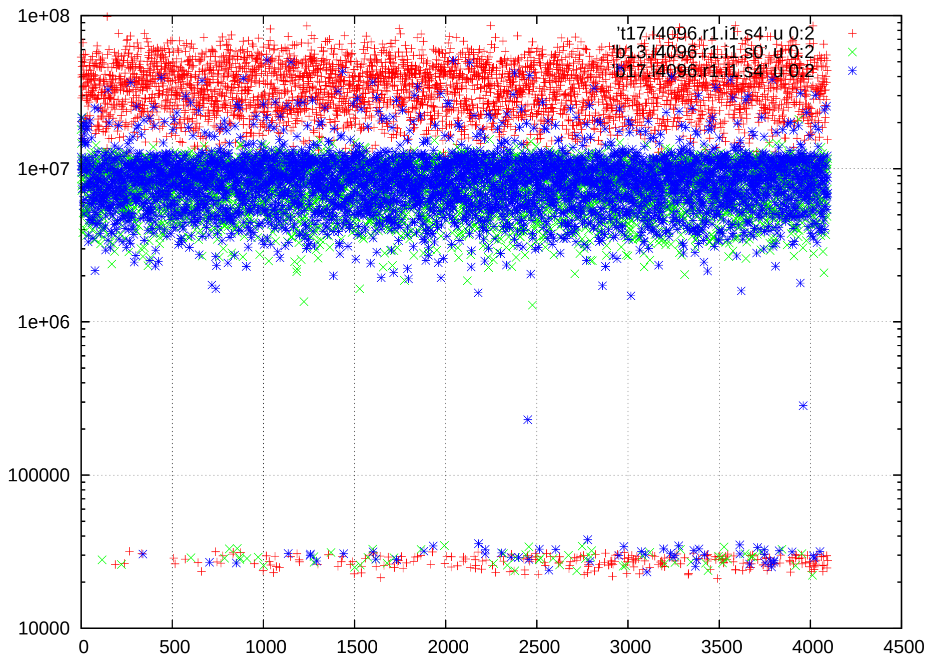
ID	Code	Non-RT	Avg.	σ	RMS
t13	Orig	–	7.80	2.77	8.28
b13	RT	–	7.72	2.77	8.20
t15	Orig	1	15.27	4.47	15.91
b15	RT	1	10.49	4.93	11.59
t17	Orig	4	33.98	15.27	37.25
b17	RT	4	8.66	4.49	9.76

2.2.2 Graph comparing RT with original

Sorted:

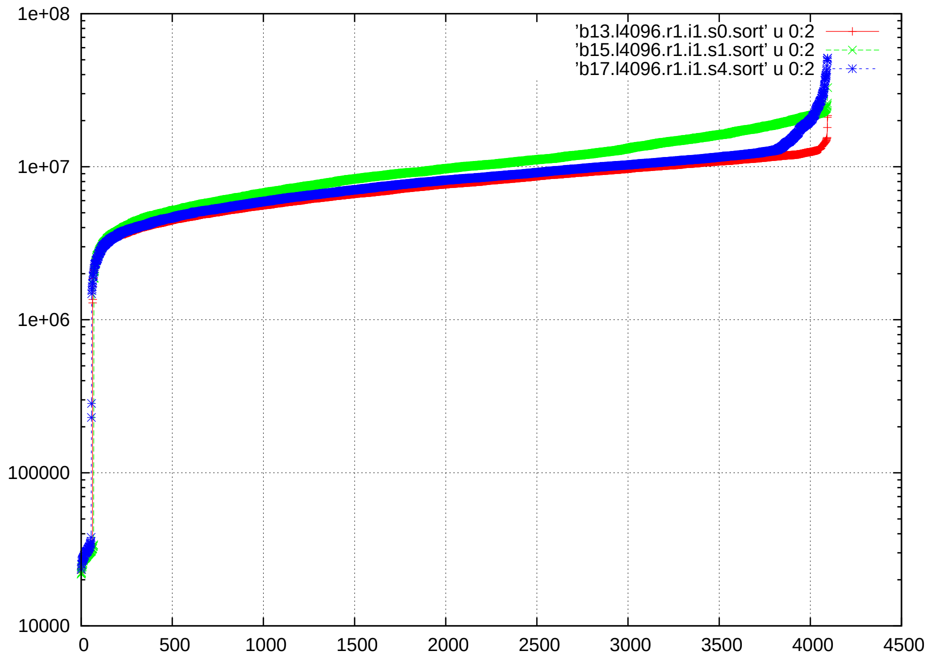


Unsorted:

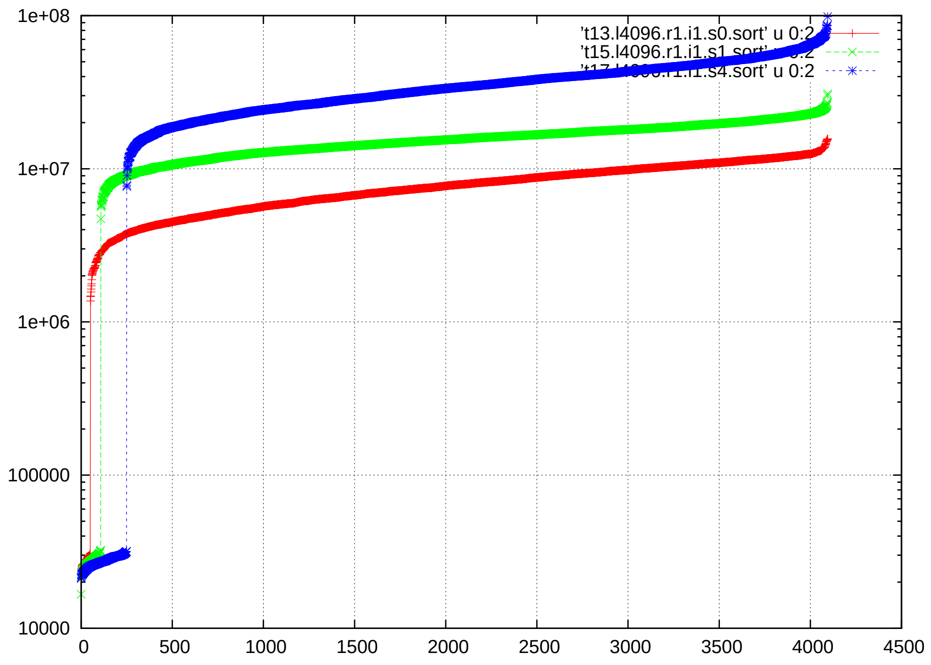


2.2.3 Graph with various number of non-RT tasks

Real-time code:



Original code:



It can be seen that the two statements hold similarly with the random dense case.

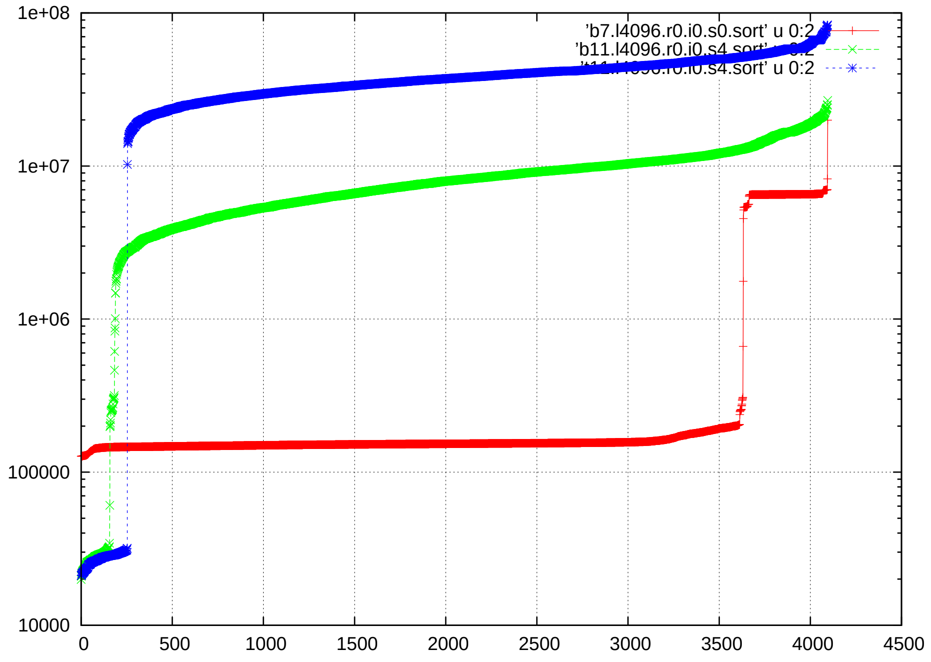
2.3 Continous Dense

2.3.1 Statistics

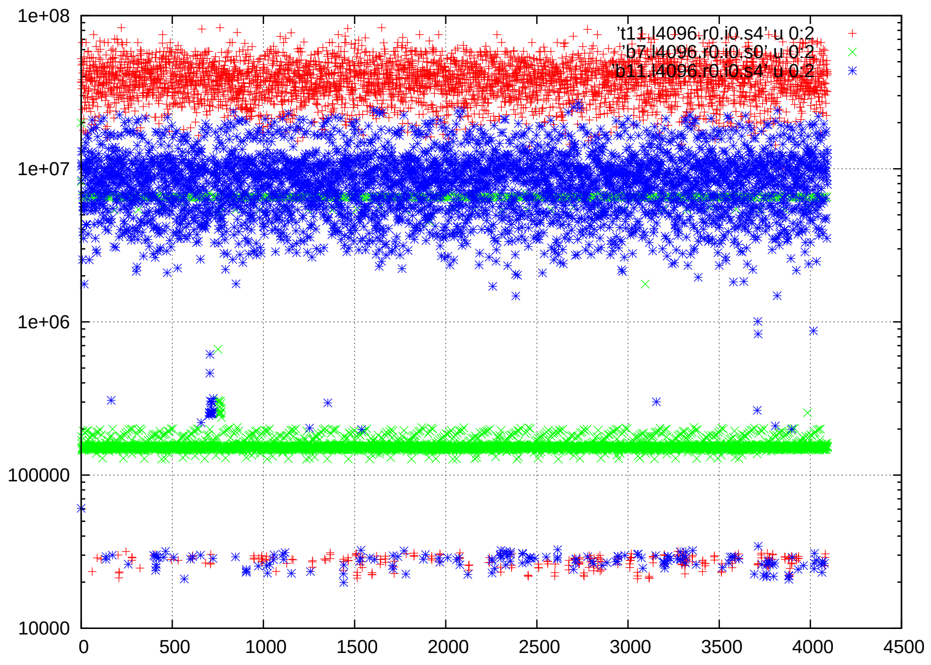
ID	Code	Non-RT	Avg.	σ	RMS
t7	Orig	–	0.87	2.05	2.23
b7	RT	–	0.87	2.03	2.21
t10	Orig	1	8.96	4.27	9.92
b10	RT	1	8.96	4.28	9.93
t11	Orig	4	36.67	14.11	39.29
b11	RT	4	8.31	4.27	9.34

2.3.2 Graph comparing RT with original

Sorted:

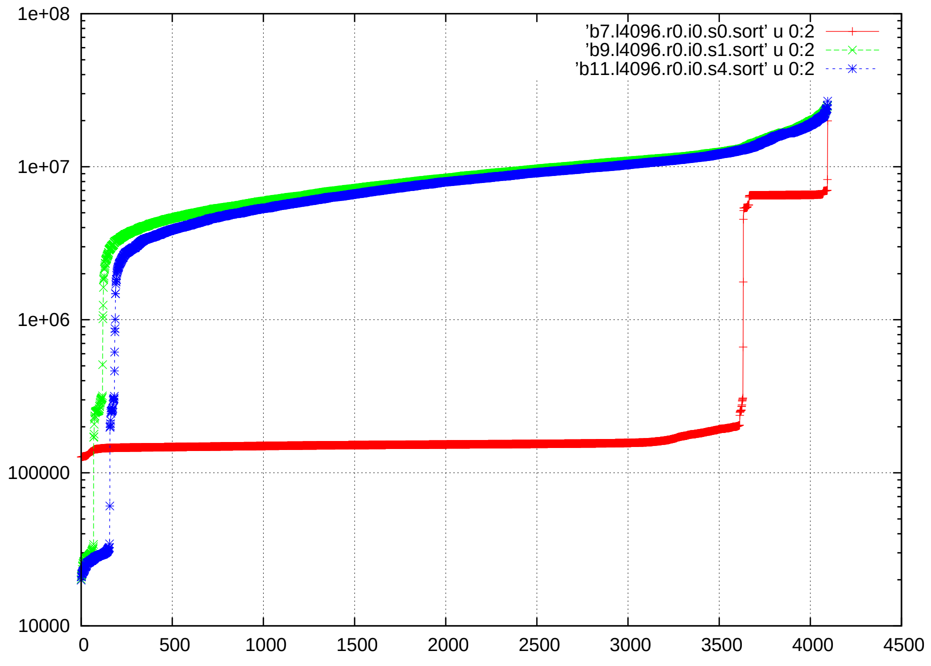


Unsorted:

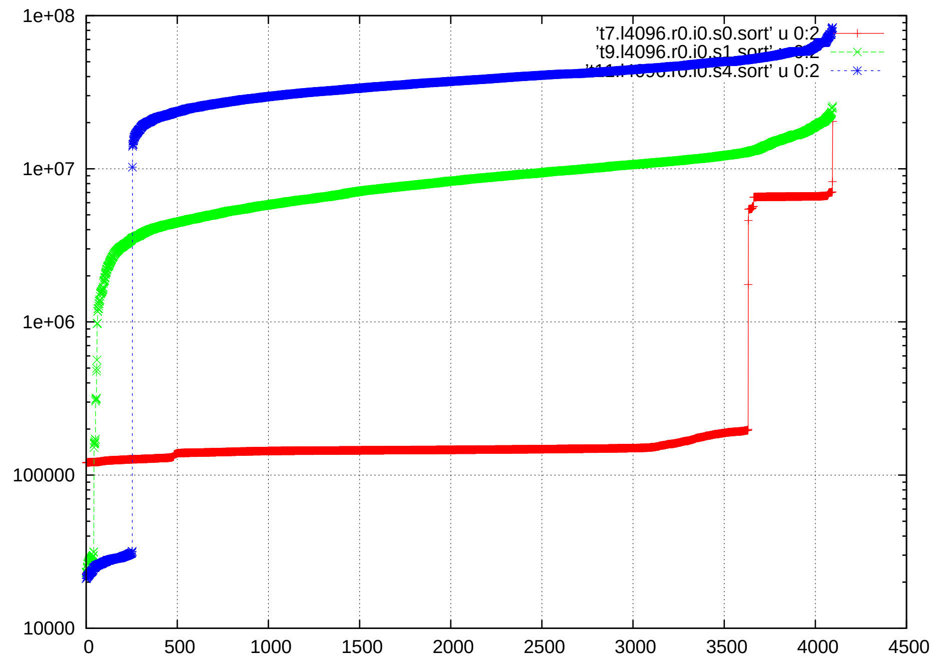


2.3.3 Graph with various number of non-RT tasks

Real-time code:



Original code:



The first statement still holds: There is much difference between original and modified code, both with non-real-time tasks.

However, the second statement does not hold: There is indeed significant difference between cases with and without non-real-time tasks, both with modified code.The reasons and possible solutions for this degradation are:

1. An unsolved priority inversion problem in Call Proxy that once a real-time request is sent to Linux kernel for processing, the Linux kernel will keep running until the request completes, which allows non-real-time tasks to run in the meantime of disk request.

   Thus non-real-time tasks get the chance to run because of the priority inversion problem above, and send requests to the request queue. This is the reason why non-real-time requests still exist when real-time requests are dense.

   In the view of real-time tasks and RTLinux kernel, the disk request becomes polling (synchronous) instead of interrupt-driven (asynchronous), where no real-time tasks can be scheduled to run while the disk is performing request.

   A possible solution to this problem: As soon as the request is sent to disk and schedule() is called in Linux kernel, schedule() gives execution to RTLinux to allow real-time tasks to execute instead of non-real-time tasks in Linux kernel. When the disk request completes and raises a interrupt, the RTLinux kernel dispatches it to Linux kernel and let it complete some post-processing of the request.

2. The nature of disk requests: Once a request is sent to disk, it cannot be canceled. At the completion of a disk request, the disk driver will select the request at the head of request queue. If there are no real-time requests in the queue, non-real-time requests will be sent to disk.

Although a real-time request might come immediately after the command is sent to disk controller, it cannot override the command being executed, thus the real-time request has to wait until the non-real-time request completes.

A possible solution to this problem: Wait a small period of time after a request completes if the head of request queue is a non-real-time request, in case a real-time request comes in the waiting period. Because real-time tasks have high priority, it can be expected that a real-time request can be selected to serve ahead of non-real-time requests in the waiting period.

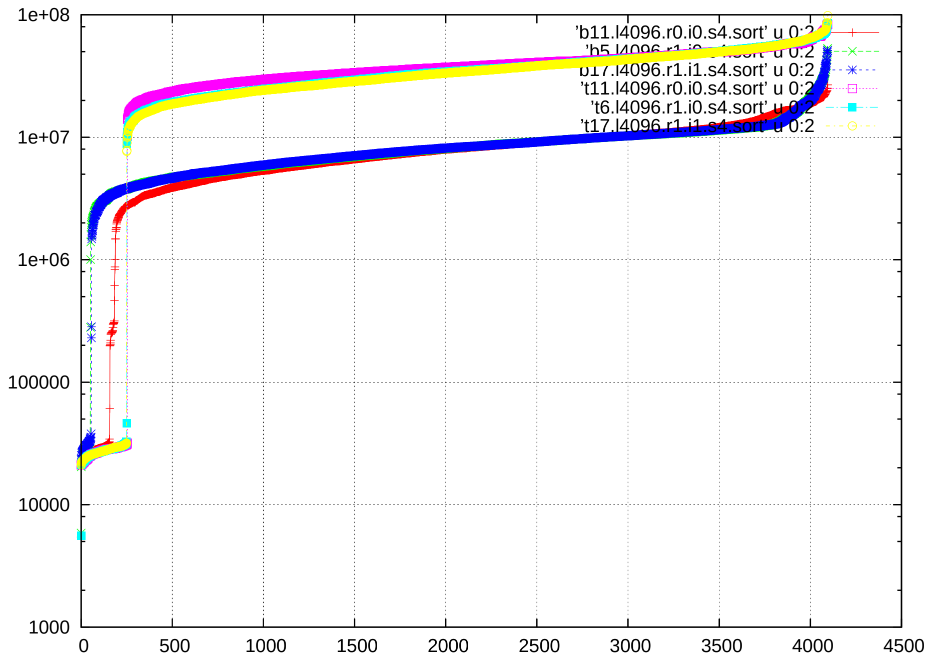
2.4 Comparison of Three Cases

2.4.1 Statistics

ID	Code	Non-RT	Avg.	σ	RMS
t11	Orig	4	36.67	14.11	39.29
t6	Orig	4	34.47	15.15	37.66
t17	Orig	4	33.98	15.27	37.25
b11	RT	4	8.31	4.27	9.34
b5	RT	4	8.63	4.48	9.72
b17	RT	4	8.66	4.49	9.76

2.4.2 Sorted Graph

Sorted graph of 6 cases shown above:



Under the same amount of non-real-time tasks, no matter what type of requests the system is loaded (continuous, random dense or random sparse), it is clear that the real-time disk system outperforms the original system significantly.

3 Conclusion

In continuous, random dense and random sparse cases, the following statements are validated:

1. There is no significent different between original and modified code, both without non-real-time tasks.
2. There is much difference between original and modified code, both with non-real-time tasks.
3. Once there are non-real-time tasks, there is no significant difference among various number of non-real-time tasks.

In random dense and random sparse cases, the following statement is validated:

- There is no significant difference between cases with and without non-real-time tasks, both with modified code.

Since the benchmark cases represent most cases in practice (see Design Report for explanation), the conclusion can be reached as follows:

1. The real-time disk file system outperforms the original disk file system in Linux 2.4 Kernel significantly in various real-time scenarios.
2. The real-time disk file system is able to distinguish real-time and non-real-time requests, while there are still space for improvement.

4 Appendix

4.1 Glossary of Statistical Data

4.1.1 Explanation for names of test cases

The name of each test case is in the form of

```
[b|t]{$2}.l{$3}.r{$4}.i{$5}.s[.sort]
```

1. *b* stands for the real-time disk system, *t* stands for the original disk system.
2. The number *1,2,...,18* together with *b* or *t* uniquely identifies a test case. Note that *t5* does not exist, since the test failed when it comes to 3980th request.
3. *l4096* stands for the number of requests. In all test cases, the request number is 4096.
4. *r0* stands for continous read, *r1* stands for random read.
5. *i0* stands for dense requests, *i1* stands for sparse requests.
6. *s0*, *s1* or *s4* stands for the number of non-real-time tasks. When it comes to *s0*, there is one real-time task running.
7. *sort* means that the test result is sorted by response time for processing. This variable need not be taken into consideration.

4.1.2 Statistics of all test cases (in nanoseconds)

The meanings of statistics have been shown in 1.3.

Test Case	Avg.	σ	RMS
b10.l4096.r0.i0.s1.sort	8958549	4284358	9930097
b11.l4096.r0.i0.s4.sort	8310869	4268811	9342850
b12.l4096.r0.i0.s4.sort	8456269	4290839	9482367
b13.l4096.r1.i1.s0.sort	7717518	2771569	8199988
b14.l4096.r1.i1.s0.sort	7720838	2726506	8188001
b15.l4096.r1.i1.s1.sort	10485931	4932294	11587767
b16.l4096.r1.i1.s1.sort	10537428	4974785	11652460
b17.l4096.r1.i1.s4.sort	8660569	4494130	9756933
b18.l4096.r1.i1.s4.sort	8718634	4713805	9911060
b1.l4096.r1.i0.s0.sort	8281162	3346846	8931757
b2.l4096.r1.i0.s0.sort	7742043	2803996	8234057
b3.l4096.r1.i0.s1.sort	10649156	4909209	11725996
b4.l4096.r1.i0.s1.sort	10582177	4986803	11698060
b5.l4096.r1.i0.s4.sort	8629315	4477794	9721668
b6.l4096.r1.i0.s4.sort	8653402	4785006	9887976
b7.l4096.r0.i0.s0.sort	874318	2025768	2206165
b8.l4096.r0.i0.s0.sort	874315	2025730	2206129
b9.l4096.r0.i0.s1.sort	8876153	4188795	9814674
t10.l4096.r1.i0.s1.sort	8956250	4268262	9921089
t11.l4096.r0.i0.s4.sort	36667964	14114900	39290219
t12.l4096.r0.i0.s4.sort	35927916	15048427	38951444
t13.l4096.r1.i1.s0.sort	7803691	2769971	8280607
t14.l4096.r1.i1.s0.sort	7747626	2827748	8247419
t15.l4096.r1.i1.s1.sort	15274692	4466165	15914081
t16.l4096.r1.i1.s1.sort	15224352	4508888	15877846
t17.l4096.r1.i1.s4.sort	33979217	15272133	37252762
t18.l4096.r1.i1.s4.sort	34375321	15189505	37580945
t1.l4096.r1.i0.s0.sort	7720288	2827631	8221702
t2.l4096.r1.i0.s0.sort	7774163	2845496	8278434
t3.l4096.r1.i0.s1.sort	15317436	4471532	15956615
t4.l4096.r1.i0.s1.sort	15341637	4489045	15984756
t6.l4096.r1.i0.s4.sort	34473818	15152361	37656103
t7.l4096.r0.i0.s0.sort	873920	2048465	2226863
t8.l4096.r0.i0.s0.sort	871994	2029326	2208513
t9.l4096.r0.i0.s1.sort	8701915	4015448	9583487

4.1.3 Statistics of all test cases (rounded to microseconds)

The meanings of statistics have been shown in 1.3.

Test Case	Avg.	σ	RMS
b10.l4096.r0.i0.s1.sort	8.96	4.28	9.93
b11.l4096.r0.i0.s4.sort	8.31	4.27	9.34
b12.l4096.r0.i0.s4.sort	8.46	4.29	9.48
b13.l4096.r1.i1.s0.sort	7.72	2.77	8.20
b14.l4096.r1.i1.s0.sort	7.72	2.73	8.19
b15.l4096.r1.i1.s1.sort	10.49	4.93	11.59
b16.l4096.r1.i1.s1.sort	10.54	4.97	11.65
b17.l4096.r1.i1.s4.sort	8.66	4.49	9.76
b18.l4096.r1.i1.s4.sort	8.72	4.71	9.91
b1.l4096.r1.i0.s0.sort	8.28	3.35	8.93
b2.l4096.r1.i0.s0.sort	7.74	2.80	8.23
b3.l4096.r1.i0.s1.sort	10.65	4.91	11.73
b4.l4096.r1.i0.s1.sort	10.58	4.99	11.70
b5.l4096.r1.i0.s4.sort	8.63	4.48	9.72
b6.l4096.r1.i0.s4.sort	8.65	4.79	9.89
b7.l4096.r0.i0.s0.sort	0.87	2.03	2.21
b8.l4096.r0.i0.s0.sort	0.87	2.03	2.21
b9.l4096.r0.i0.s1.sort	8.88	4.19	9.81
t10.l4096.r1.i0.s1.sort	8.96	4.27	9.92
t11.l4096.r0.i0.s4.sort	36.67	14.11	39.29
t12.l4096.r0.i0.s4.sort	35.93	15.05	38.95
t13.l4096.r1.i1.s0.sort	7.80	2.77	8.28
t14.l4096.r1.i1.s0.sort	7.75	2.83	8.25
t15.l4096.r1.i1.s1.sort	15.27	4.47	15.91
t16.l4096.r1.i1.s1.sort	15.22	4.51	15.88
t17.l4096.r1.i1.s4.sort	33.98	15.27	37.25
t18.l4096.r1.i1.s4.sort	34.38	15.19	37.58
t1.l4096.r1.i0.s0.sort	7.72	2.83	8.22
t2.l4096.r1.i0.s0.sort	7.77	2.85	8.28
t3.l4096.r1.i0.s1.sort	15.32	4.47	15.96
t4.l4096.r1.i0.s1.sort	15.34	4.49	15.98
t6.l4096.r1.i0.s4.sort	34.47	15.15	37.66
t7.l4096.r0.i0.s0.sort	0.87	2.05	2.23
t8.l4096.r0.i0.s0.sort	0.87	2.03	2.21
t9.l4096.r0.i0.s1.sort	8.70	4.02	9.58