Characterizing TPC-H on a Clustered Database Engine from the OS Perspective

Yanyong Zhang[†], Jianyong Zhang[†], Anand Sivasubramaniam[†], Chun Liu[†], Hubertus Franke[‡]

[†] Department of Computer Science & Engineering
The Pennsylvania State University
University Park PA 16802
{yyzhang, jzhang, anand, chliu}@cse.psu.edu

[‡] IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights NY 10598-0218
{frankeh}@us.ibm.com

Abstract

A range of database services are being offered on clusters of workstations today to meet the demanding needs of applications with voluminous datasets, high computational and I/O requirements and a large number of users. The underlying database engine runs on cost-effective off-the-shelf hardware and software components that may not really be tailored/tuned for these applications. At the same time, many of these databases have legacy codes that may not be easy to modulate based on the evolving capabilities and limitations of clusters. An indepth understanding of the interaction between these database engines and the underlying operating system (OS) can identify a set of characteristics that would be extremely valuable for future research on systems support for these environments. To our knowledge, there is no prior work that has embarked on such a characterization for a clustered database server.

Using a public domain version of a commercial clustered database server and TPC-H like[1] decision support queries, this paper studies numerous issues by evaluating performance on an off-the-shelf Pentium/Linux cluster connected by Myrinet. The execution profile clearly demonstrates the dominance of the I/O subsystem in the execution, and the importance of the communication subsystem for cluster scalability. In addition to quantifying their importance, this paper provides further details on how these subsystems are exercised by the database engine in terms of characteristics such as request sizes, spatial and temporal distributions. These characteristics provide insight on the benefits of possible optimizations in these subsystems. This includes the potential savings by avoiding copies across protection domains during I/O and the potential reduction in the number of messages by employing multicasts. Mechanisms for performing such optimizations are also discussed.

1 Introductions

Clusters of workstations built with commodity processing engines, networks and operating systems are becoming the platform of choice for numerous high performance computing environments. Commodity hardware and software components make the price, upgradeability and accessibility of clusters very attractive, facilitating their widespread deployment in several application domains.

The speed at which clusters are being deployed for these diverse and challenging environments is outpacing the rate of progress in the systems software technologies and tools that are crucial building blocks for the applications. Most commodity off-the-shelf software (including the operating system) are not specifically tuned for cluster environments, and it is not clear if gluing together individual operating systems, that do not know the presence of each other, is the best approach to handle such loads. Further, off-the-shelf operating systems are meant to be for general purpose usage, with most of them really tuned for desktop applications or uniprocessor/SMP class server applications. Their suitability for cluster applications is not well understood. At the same time, one does not want to design/develop operating systems specifically for clusters, which would then go against the off-the-shelf rationale.

Just as many of today's operating systems (such as Linux) are not specifically customized for these emerging (some of these - like the database engines - are not really new, but are clustered implementations of the original version) applications on clusters, the applications are in turn not extensively tuned for these operating systems. An important reason is the fact that some of these, at least the database engines, have legacy codes that have evolved over several revisions/optimizations over the years, and it is difficult to fundamentally change their design overnight in light of these new systems (regardless of how modular they may be), which are still evolving. There is a substantial cost that is expended in testing/debugging to write specific software that exploits special features of the underlying system, and it is not clear if the ensuing rewards can offset this cost. The clustered versions of these legacy applications can be viewed more as an

[1] These results have not been audited by the Transaction Processing Performance Council and should be denoted as "TPC-H like" workload.

exercise in porting, taking into account the new technologies and capabilities/limitations offered by the clusters. One cannot blame these developments since commercial vendors are often pressurized by several factors to get a product out of the door for a target platform as early as possible, and there is a need to support the product on several different platforms. Consequently, we have many legacy applications, such as the database engines which are the focus of this paper, running on operating systems that were not the initial targets of their implementation and on clusters which were not the initial target hardware. Over a period of time, revisions/improvements to the products are likely to address such concerns. There is visible evidence of this in the fact that there are ongoing research activities [3] from vendors, who already have commercial offerings, exploring alternate implementation styles

It is unavoidable to encounter such situations when technologies change, and it is unclear whether the application needs to be tailored/tuned for the underlying OS on a cluster, or vice-versa, or if we need to do a combination of the two. With the large code base of many of these legacy database engines, one could hypothesize that it would be easier to fine tune the operating system, especially with an open source OS such as Linux. A lot of work has already gone into optimizing the legacy applications, and the issues/optimizations may not be very different even for these new environments/OS.

This leads us to believe that there is the possibility of a middle ground, wherein if we know what issues are really important, then we could incorporate a few mechanisms/extensions in the OS and with a few user/configuration directives (or even slight application modifications) be able to enjoy the benefits of better matching the application with the underlying system. At the same time, such OS mechanisms/extensions may be rewarding for other applications as well and could very well become a feature of the OS in future offerings. Our goal in this paper is not to develop application-specific operating systems, nor is it to find out what OS mechanisms/capabilities are needed for extensibility/customization as other researchers have done [3]. Rather, coming from the applications viewpoint, we would like to make a list of recommendations based on the execution characteristics that can benefit future developments. We have also taken the liberty of suggesting possible mechanisms and their implementation (specifically in Linux) for optimizing the execution based on these gleaned characteristics. There are, arguably, other possible mechanisms/implementations for performing the same optimizations (even on Linux), or one could use these characteristics for customizing an extensible OS [2, 6] accordingly. Another possibility is to provide middleware that can better match the applications with the OS based on these characteristics.

In summary, a detailed characterization of the execution of applications on a cluster from the OS perspective can contribute to the knowledge-base of information that can be used for guiding future developments in systems software and applications for these environments. It would also be invaluable for fine tuning the execution for better performance and scalability, since each of these applications/environments has high commercial impact. In this paper we focus specifically on TPC-H queries, a decision-support database workload. This constitutes an important workload for business enterprises, with long running queries - ranging from a few minutes to a few days - that can benefit from the capabilities of a cluster.

It is well understood that I/O is the biggest challenge faced by database engines on uniprocessors/SMPs [7, 8, 9, 10] and there is a large body of prior work proposing hardware and software enhancements to address this problem. It is not clear if I/O becomes any less important when we move the engine to a cluster environment, since there is another factor to consider, which is the network communication. System scalability with cluster size is dependent on how parallel is the computation division across the cluster nodes, how balanced are the I/O activities on different nodes, and how does the communication traffic change with data set and cluster sizes. All this requires a careful profiling and analysis of the execution of the queries on the database engine. To our knowledge, that there has been no prior investigation of completely characterizing the execution of TPC-H on a clustered database engine, and studying these characteristics for optimization at the application-OS boundary.

Section 2 gives details on the experimental setup. Section 3 gives the overall system execution profile and the system scalability is examined in Section 4. Based on the system profile, the I/O and network characteristics and optimizations are discussed in sections 5, 6 and 7. Finally, Section 8 summarizes the results and contributions of this study.

2 Experimental Setup

TPC-H contains a sequence of 22 queries (Q1 to Q22), that are fired one after another to the database engine [2]. In this work, we consider response time for each query as main measure, i.e. the time interval between submitting the query and getting back the results.

All the tables are horizontally partitioned across the entire cluster using a hash-based scheme, and we have verified that this results in a balanced distribution across nodes. There is a client machine (not part of the cluster) that sends these queries to a database coordinator node on the cluster, which then distributes the work and gives back the results to the client. Each node of the cluster performs the queries on the rows residing on it and exchanges results via Myrinet if necessary. The client is connected to the cluster using Myrinet. As was mentioned, we run experiments on an 8 dual node Linux/Pentium cluster, that has 256 MB RAM and 18 GB disk on each node. The nodes are connected by both switched Myrinet and Ethernet, and we study these networks separately. We use Linux 2.4.8, which was the latest release at the time of conducting

[2] Q21 and Q22 take an inordinately long time and the results for these two queries are not included.

the experiments. (Please note that up to version 2.4.8. standard Linux kernels do not support raw disk IO interfaces.) This kernel has been instrumented in detail to glean different statistics, and also modified to provide insight on the database engine execution since we are treating it as a black box. We have also considered the overheads of instrumentation by comparing the results with those provided by the proc file system to ensure validity of what is presented here.Unless otherwise stated, the experiments use kernel level TCP over Myrinet for communication, and the dataset is 30 GB in size.

3 Operating System Profile

We first present a set of results that depict the overall system behavior at a glance. The following results have been obtained by both sampling the statistics exposed by the Linux proc file system (`stat`, `net/dev`, `process/stat`) as well as by instrumenting the kernel. The kernel instrumentation was done by inserting code in the Linux system call jump mechanism, as well as in the scheduler and points where there is pre-emption (such as blocking) or resumption. The proc file system information is used to present the percentage utilization of the system in different modes, the rates/frequency of I/O, page fault and network activities. The profile of different system calls is presented from the kernel instrumentation.

The results are shown in Table 1, which gives system statistics for each query in terms of: the percentage of time that the query spent executing on the CPUs in user mode (relative to its overall execution time), the percentage of time that the query spent executing on the CPUs in system mode (relative to its overall execution time), the average number of page faults incurred in its execution per jiffy (10 milliseconds in Linux), average number of file blocks read per jiffy, average number of file blocks written per jiffy, average number of packets sent over the network per jiffy, the average number of packets received from the network per jiffy, and the percentage utilization of the CPU(s) by the database engine during I/O operations (captures the overlap of work with I/O operations). The file block size is 4096 bytes, and the Maximum Transfer Unit (MTU) for network packets is 3752 bytes. In addition to these, the table also shows the top four system calls (in terms of time) exercised by each query during its execution, and the percentage of system time that is spent in each of these calls. These statistics help us understand what components of the OS are really being exercised, and the relative importance of these components.

From these results, we make the following observations:

- As is to be expected with database applications, the bulk of the execution time in the system mode is taken up by file system operations (pread/pwrite). These calls are employed to read and write the queried relational tables, as well as for any temporary tables that are needed along the way. Our examination of the execution leads us to believe that the considered database server goes via the file system for I/O accesses, and does not directly use raw disks or mmap operations.

  Disk operations are so dominating in some queries (Q1, Q8, Q12, Q17) that the CPU utilization does not cross 50% in these queries. I/O costs not only result in poor CPU utilization overall (because of waiting for disk operations to complete), but also in significantly increasing the system call overhead itself. Note that this system call overhead (system CPU time) does not include the disk latencies. Rather, this high overhead is due to memory copying, buffer space management and other bookkeeping activities. In some cases (such as Q12), the system CPU time (overheads) even exceeds the amount of time spent executing the useful work in the query at the user-level.

- Most of the I/O that is incurred is more due to reads than write operations. This is particularly characteristic of TPC-H queries, because most operations are for decision-support (requiring only reads).

- Though the numbers are not explicitly given here, we would like to point out that the high read overheads are not only because of the higher number of file system read calls, but are also due to the higher cost per invocation of this call. We noticed that a pread call can run to nearly a millisecond in some queries.Of all the system calls considered, we found the per pread invocation taking the maximum amount of time.

- When we examine the CPU utilization during I/O (last column of Table 1), we find that there is good overlap of work with disk activity in some queries. As we will point out later on in this paper, the bulk of I/O is initiated by the database prefetcher, which does not necessarily come into the critical path of the execution in many queries. However, queries such as Q12, encounter significant blocking.

- After the file system calls, we found socket calls (select, socketcall) to be the next dominant system overhead.

- Interprocess communication (IPC), though not as dominant as the other two OS components, does come in third in the overall system overheads.

- Despite the dominance of I/O in many queries, queries like Q11 have a high CPU fraction (particularly in the user mode). Even though there are I/O operations in these queries, their costs are overshadowed by useful work (CPU utilization is around 66% even during periods of disk activity, and the bulk of it is in the user mode). Another point to note from this observation, and in the fact that there is little variation in these results from node to node, is the hypothesis that such queries are likely to scale very well as we move to larger clusters since they can benefit from higher degrees of parallelism.

query	user CPU (%)	system CPU (%)	system CPU breakup (%)				page faults per jiffy	blocks read per jiffy	blocks written per jiffy	packets sent per jiffy	packets received per jiffy	CPU utilization during IO (%)
Q1	27.58	21.44	pread 46.7	pwrite 46.7	select 3.3	ipc 2.9	1.50	51.01	23.1151	0.0012	0.0015	26.87
Q2	56.22	15.67	socketcall 35.4	pread 32.9	select 20.1	pwrite 7.3	0.39	21.97	1.7718	1.9077	1.9248	53.73
Q3	40.76	17.76	pread 54.1	socketcall 15.1	select 13.1	pwrite 12.8	0.99	55.47	4.9591	0.9778	0.9938	55.40
Q4	51.48	15.19	pread 60.0	select 17.6	socketcall 11.2	pwrite 6.9	0.00	22.97	1.0478	0.3517	0.3652	68.41
Q5	58.96	15.68	socketcall 43.3	pread 29.2	select 21.7	pwrite 3.7	0.05	16.73	1.1369	2.0779	2.0849	42.81
Q6	40.65	22.20	pread 90.1	ipc 4.9	socketcall 4.3		1.73	90.72	0.0020	0.0012	0.0012	33.49
Q7	52.44	16.82	pread 72.1	pwrite 14.8	ipc 6.1	select 6.0	0.00	16.89	1.9467	2.3880	2.3521	31.04
Q8	20.65	17.71	pread 59.5	pwrite 23.7	select 9.4	ipc 5.8	0.01	27.63	4.8078	0.0261	0.0228	12.91
Q9	51.41	13.52	pwrite 40.8	pread 38.7	select 13.9	ipc 2.5	0.00	6.69	1.9276	0.0133	0.0136	23.21
Q10	41.79	17.87	pread 57.7	socketcall 17.9	select 13.4	pwrite 7.0	0.17	41.99	1.8880	0.8774	0.8859	18.71
Q11	81.00	13.28	socketcall 43.6	pread 27.0	select 24.8	ipc 3.9	0.49	18.87	0.0020	2.3794	2.4011	66.46
Q12	14.91	19.73	pread 78.3	selet 15.2	ipc 5.2		0.25	40.79	0.0197	0.0102	0.0101	4.8
Q13	53.23	21.86	pread 45.7	socketcall 30.3	select 18.8	ipc 4.6	1.62	45.86	0.0034	2.0966	2.0935	32.71
Q14	33.57	22.19	pread 85.6	select 8.3	ipc 5.1		1.01	84.78	0.0025	0.1156	0.1175	29.75
Q15	55.37	18.69	pread 71.7	select 14.0	ipc 10.9	nanosleep 1.9	0.60	75.26	0.0033	0.6260	0.7703	51.68
Q16	51.84	15.30	socketcall 43.5	pread 27.0	select 22.2	ipc 5.7	2.32	15.36	0.8454	2.4836	2.4993	46.24
Q17	23.71	18.26	pread 59.4	pwrite 26.1	select 8.7	ipc 4.9	0.00	32.76	5.2532	0.0036	0.0037	13.94
Q18	52.64	14.77	pread 61.1	socketcall 17.3	select 13.3	pwrite 5.2	0.04	17.16	0.6261	0.3890	0.3803	35.11
Q19	35.48	21.16	pread 80.1	select 7.3	ipc 5.9	socketcall 5.9	0.29	78.76	0.0032	0.3343	0.3329	23.38
Q20	56.98	14.72	pread 53.4	select 25.6	socketcall 15.3	ipc 3.6	0.00	15.74	0.1601	0.3456	0.2973	47.36

Table 1: System Profile (statistics are collected from node 1)

4 System and Workload Scalability

The previous experiments used the 8-node dual configuration on a 30 GB dataset, with Myrinet as the interconnect. We briefly present results to discuss the scalability of the execution as a function of number of nodes, the network (switched 1.26 Gbps Myrinet vs 100 Mbps Ethernet), and to study the trade-offs between a larger cluster vs a smaller one with more processors at each node (compare 4 node duals with 8 node uniprocessors), in Figure 1 for a 1 GB dataset. Results are normalized with respect to the 8 node dual configuration. It should be noted that a complete scalability study across the spectrum of parameters is well beyond the scope of this paper. Rather we are only trying to point out the relative performance and issues to understand the implications of some of these parameters.

We observe that increasing the number of nodes has several advantages. First, you get higher parallelism in computation and in I/O. The other significant benefit is in the ability to harness more physical memory (buffer space) across the nodes for the same dataset [16], which can reduce I/O further. The downside is the additional communication that may be incurred. We find that the benefits outweigh the drawbacks for nearly all queries except Q2 and Q17, even when we move from 1 to 2 nodes. This can be explained by the fact that Q2 has higher communication and Q17 has much lower CPU utilization that can benefit from parallelism (see Table 1). Moving on to 4 or 8 nodes, we find that in Q9, Q10, Q11, and Q20, overheads from parallelism hurt their performance as well. In nearly all other queries, we find significant savings (particularly at the smaller cluster sizes) from parallelism. A large portion of this savings is due to the higher buffer availability (reducing the total I/O, and not just providing parallel accesses to disks), that results in a superlinear performance improvement for queries such as Q9, Q11, Q12, Q15, and Q20. In general, we observe that 4 and 8 dual node configurations are good operating points for this dataset size of those considered. After a certain size, the savings from larger memory drop a little, with the overheads also offsetting a large portion of these savings and the gain in parallelism.

Next, when we compare the 4 node dual with the 8 uniprocessor nodes, we find that there is no clear winner. In Q1, Q3, Q4, Q5, Q6, Q8, Q12, Q14, Q17, Q18, and Q19, the uniprocessor node configuration is better, and the 4 node dual is better for the others. If you look at these queries in Table 1, these

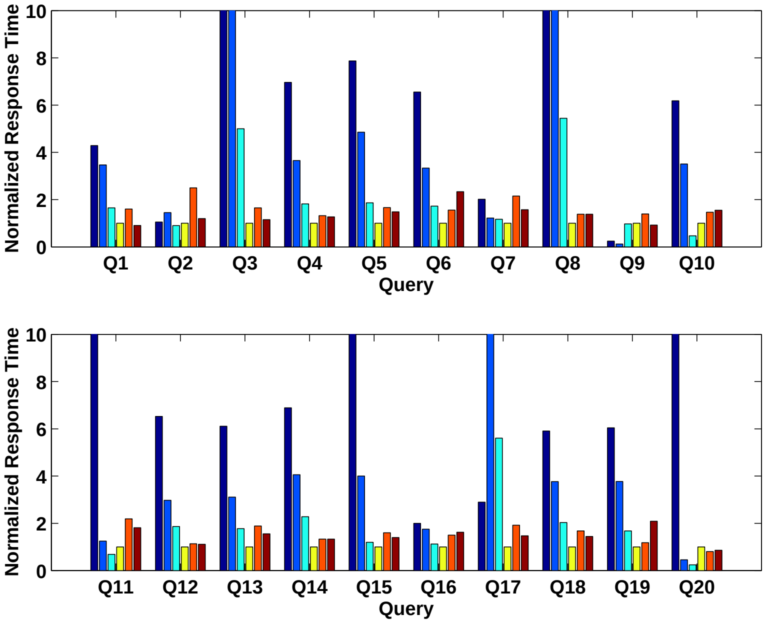


Figure 1: For each query, there are six bars (from left to right) showing the query response (execution) time under the following six configurations respectively: (i) one 2-way SMP node; (ii) two 2-way SMP nodes, myrinet; (iii) four 2-way SMP nodes, myrinet; (iv) eight 2-way SMP nodes, myrinet; (v) eight uniprocessor nodes, myrinet; and (vi) eight 2-way SMP node, 100Mbps Ethernet. We normalize the execution times under each configuration with respect to the time under configuration (iv). The dataset under examination is 1G. Note that the bars which reach the top of the graph actually exceed 10, and they are chopped in order to better show the differences between others.

have much lower CPU utilization, suggesting that disk activity is the bottleneck here. Even though both configurations provide the same number of CPUs, the 8 node configuration provides higher disk parallelism that benefits these queries. In the others, the communication is higher, and that becomes much more of a problem with a small 1 GB database considered in these experiments than the I/O parallelism.

It should be noted that in all these experiments we have used *constant problem size scaling* [4], with the physical memory increasing as we increase the cluster size. It would also be interesting to explore memory constrained scaling, or to consider a smaller cluster with more memory per node compared to a larger cluster with less memory per node (say keeping the overall cluster cost in dollars the same). Such issues are beyond the scope of this paper and we intend to explore these in the future.

When we move from a slow network (Ethernet) to a fast switched Myrinet platform, on the average query execution gets speeded up by 25%. We are not explicitly presenting results varying different dataset sizes, though we have conducted those experiments. In general, a larger dataset scales better as a function of the cluster size.

Before concluding this section, we would like to reiterate the importance of I/O and communication as we move to the future. Definitely, communication becomes important with larger clusters. At the same time, I/O can benefit significantly from such clusters not only because of the parallelism to disks, but also because of the higher memory capacities. Still, the I/O bottleneck would continue to pose challenges for clustered database services as dataset sizes increase. Further, I/O and communication become all that much more prominent with faster CPUs, and these are the focus of our attention in the rest of this paper.

5 I/O Subsystem: Characterization and Possible Optimizations

The results from the system profile clearly illustrate the importance of I/O for database servers. We now set out to look at the I/O subsystem more closely, trying to characterize its execution and look for possible optimizations.

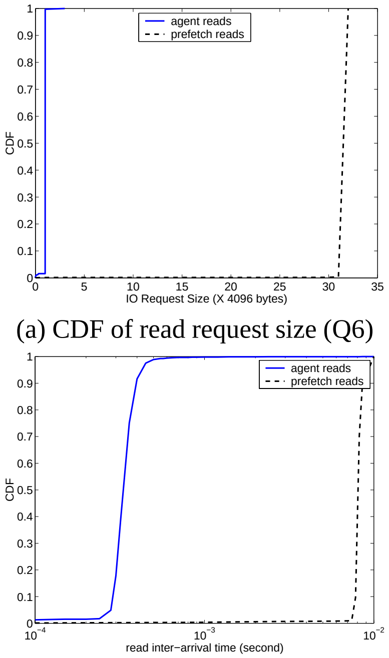
5.1 Characteristics

Table 2 sorts the queries in decreasing order based on the fraction of total query execution time spent in the pread system call obtained from earlier profile results. We can see that pread is a significant portion of the execution time in many queries. It takes over 10% of the execution time in 11 of the queries. It should be noted that this is the time spent in the system call (i.e. in buffer management, book-keeping, copying across protections domains, etc.), and does not include the disk costs itself. This implies that it is *not only important*

Query	Q6	Q14	Q19	Q12	Q15	Q7	Q17	Q8	Q10	Q1
% of exec. time	20.0	19.0	16.9	15.4	13.4	12.1	10.8	10.5	10.3	10.0
Query	Q13	Q3	Q4	Q18	Q20	Q2	Q9	Q5	Q16	Q11
% of exec. time	10.0	9.6	9.1	9.0	7.9	5.2	5.2	4.6	4.1	3.5

Table 2: pread as a percentage of total execution time

to lower or hide disk access costs, but to optimize the pread system call itself. In the interest of space, We focus on query Q6 which incurs the maximum pread overhead in the rest of this section. (the trends/arguments are similar for the others). Further, as with the profile results, we did not observe much variation across the nodes, and consequently examine the executions from the viewpoint of each node.



(a) CDF of read request size (Q6)

(b) CDF of inter-read time (Q6)

Figure 2: IO characterization results in terms of read request size and time between successive reads.

First, we would like to briefly explain how we believe the reads are invoked in the query execution. DB2 has several agent processes that actually perform the work in the queries, and prefetcher processes that make pread calls to bring in data from disk ahead of when the agent may need it. Figure 2 shows that the agent reads are more bursty, coming in closer proximity, than the reads issued by the prefetcher.

We found that the preads issued by the agent are usually for a block that has been recently read by the prefetcher just before that invocation. It may come as a surprise as to why this block could not have been serviced by the prefetcher directly (if it was only recently read), instead of going to the kernel. One possible explanation is that the agent is doing a write on this block, and it may not want to write into a page that is residing in the prefetcher. Instead of making instead of making ipc calls to remove the page from the prefetcher, it would be better to create a copy within the agent by using a pread call directly.

For further credibility on this hypothesis, before returning from pread calls, we modified the kernel to set the corresponding data pages to be read-only mode, and we found the agent to incur (write) segmentation faults (indicated as copy-on-write in Table 4) on nearly all those pages (compare the copy-to-user and copy-on-write columns for the agent in Table 4). Finally, it should be noted that the agent pread calls are much lower (both in terms of the number of calls and in terms of the number of blocks read) than those for the prefetcher.

We also include in Table 3 the fraction of pread block requests that hit in the Linux file cache for the prefetcher and the agents. As was pointed out, the agent requests come very soon after the prefetcher request for the same block, and thus nearly always hit in the Linux file cache. With the prefetcher requests on the other hand, we find the file cache hits range between 40-60%. We mentioned earlier that the prefetcher requests are usually for 32 blocks at a time. The Linux file cache manager itself does some read ahead optimizations based on application behavior and brings in 64 blocks (twice this size). With a lot of regularity (sequentiality) in I/O request behavior for this workload, this read ahead tends to cut down the number of disk accesses by around 50%.

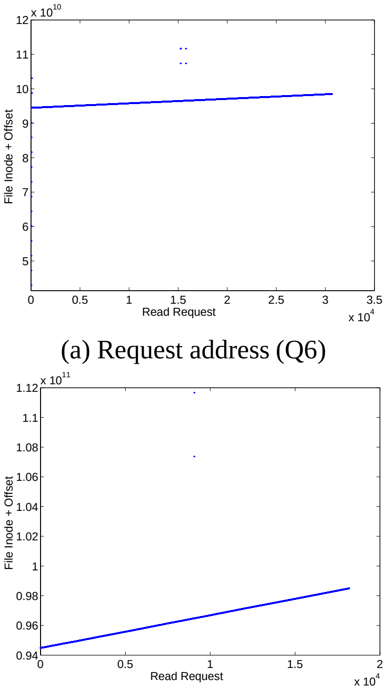
One can observe this regularity or patterns in the I/O request blocks by looking at Figure 3 (a) which shows the block number (expressed as a combination of inode + block number within file) requests that are issued to system. Further, Figure 3 (b) gives the same information for those requests that miss in the Linux file cache (on the average, every alternate request from (a) would miss here). One can visually observe regularity/sequentiality in both the requests that are generated and in the addresses that miss in the file cache. Prefetching and read-ahead are thus extremely useful for these executions.

5.2 Recommendations and Possible Optimizations

The previous subsection showed that system overheads in preads (not in the actual disk I/O) are a significant portion of the execution time. These read calls are usually for several blocks (32) by the prefetcher (that dominate the occasional single block reads by agent), and these blocks are hardly modified (TPC-H being a decision support workload, this observation is not very surprising). Further, many of these blocks are actually present already in the Linux file cache. We next explore how we can optimize the pread calls based on this

Query	prefetcher hit ratio	agent hit ratio	Query	prefetcher hit ratio	agent hit ratio
Q1	0.5711	1.0000	Q11	0.2788	1.0000
Q2	0.5321	1.0000	Q12	0.4735	1.0000
Q3	0.5164	1.0000	Q13	0.5261	1.0000
Q4	0.4873	1.0000	Q14	0.5344	1.0000
Q5	0.5991	1.0000	Q15	0.4227	1.0000
Q6	0.4729	1.0000	Q16	0.4409	1.0000
Q7	0.5182	1.0000	Q17	0.7365	1.0000
Q8	0.5300	1.0000	Q18	0.4226	1.0000
Q9	0.4683	1.0000	Q19	0.5625	1.0000
Q10	0.5082	1.0000	Q20	0.5453	1.0000

Table 3: Fraction of block requests that hit in Linux file cache for prefetcher and agent requests



(a) Request address (Q6)

(b) Page cache miss address (Q6)

Figure 3: pread request addresses and the corresponding addresses that miss in the Linux page cache (that are sent to disk)

information.

A significant portion of pread cost is expended in copying data (that is in a block in the file cache, either already or brought in upon disk I/O completion), from the kernel file cache to a user page, which needs to cross a protection-domain boundary (using the *copy-to-user()* mechanism). In the current 2.4.8 Linux implementation, a copy is actually made at this time.

This problem of reducing copying overheads for I/O has been looked at by several previous studies [14, 5, 12]. There are different techniques one could use, and a common one is to simply set the user page table pointer to the buffer in the file cache. This could affect the semantics of the pread operation in some cases, particularly when more than one user process reads the same block. In the normal semantic, once the copy is done, a process can make updates to it without another seeing it, while the updates would be visible without copies. This is usually addressed (as is by Linux in several other situations) by the *copy-on-write* mechanism, Some studies [5, 14, 12] suggest that even this may not be very efficient since updating virtual address mappings can become as expensive as copying. Instead, sharing of buffers between user and kernel domains is advocated. In this paper, we are not trying to advocate any particular technique for reducing these copies. Rather, we would like to find out what would be the benefit of reducing copy costs.

Remember, that a copy-to-user is actually needed when the pread is not page aligned and/or is interested in only part of the page. However, from our characterization results we find that most requests are page aligned and are in fact for an integral (32) number of papers. As a result, one could use the virtual remapping approach to implement the reduction of copies in these queries.

To examine the potential benefits of such an implementation, we track the total number of copy-to-user() calls that are made (actually one for each page) and the number of these calls that cannot be avoided (you cannot avoid it when there is a write segment violation and we need to do a copy-to-user at that time), during the execution of these queries after setting these pages to read-only mode. These numbers are shown in Table 4. As we can observe, the number of copy-on-writes that are actually needed is much lower than the number of copy-to-user invocations, as was suspected initially. In general, we get no less than 65% savings in the number of copies, with actual savings greater than 80% for most queries (see the last column of this table). Most of these savings are due to the prefetcher reads. Our measurements of copy-to-user routine for a single block using the high resolution timer takes around 30 microseconds for one page. For 32 block reads that the prefetcher issues, avoiding this cost can be a significant savings. This is particularly true when the blocks hit in the file cache (and there is no disk I/O) since this cost is a significant portion of the overall time required to return back to the application. Table 3 shows that this happens nearly 50% of the time. Even with disk activity, Table 1 shows CPU utilization

Query	prefetcher			agent			total
	copy-on-write	copy-to-user	% Reduction of copies	copy-on-write	copy-to-user	% Reduction of copies	% Reduction of copies
Q1	0	1040228	100	11551	11565	1.2	98.9
Q2	0	383334	100	63145	63145	0	85.7
Q3	0	1253107	100	52155	52157	0.003	96.0
Q4	0	997507	100	235758	235759	0.0004	80.9
Q5	0	1007919	100	307689	307689	0	100.0
Q6	0	914482	100	47	47	0	100.0
Q7	0	1084454	100	276790	276791	0.0003	79.7
Q8	0	978134	100	255057	255060	0.001	79.3
Q9	0	2478154	100	316500	316502	0.0006	88.7
Q10	0	974062	100	278213	278215	0.0007	77.8
Q11	0	170643	100	6833	6834	0.01	96.1
Q12	0	911544	100	134933	134933	0	87.1
Q13	0	184491	100	42	43	2.3	100.0
Q14	0	945619	100	38429	38430	0.003	96.1
Q15	0	1175166	100	38394	38395	0.003	96.8
Q16	0	36137	100	15001	15003	0.01	70.7
Q17	0	1968122	100	113962	113963	0.0008	94.5
Q18	0	1945777	100	502	503	0.19	100.0
Q19	0	865529	100	38429	38429	0	95.7
Q20	0	847755	100	50442	50444	0.003	94.4

Table 4: % of copy-to-user calls that can be avoided. Of the given copy-to-user calls, only the number shown under the copy-on-write are actually needed. The statistics are given for the prefetcher and agents separately, as well as the overall savings.

higher than 50% in most queries, suggesting that removing this burden of copying by the CPU would help query execution.

There is a caveat that we would like to point out with respect to the page remapping solution for reducing copying costs particularly with this database workload. With the prefetcher being quite active, and getting pages that are very often found in the Linux file cache, there is the possibility of very soon having a number of virtual address mappings to the file cache buffers (rather than to the buffers in the prefetcher itself). The file cache would then have to be made much larger (the buffer manager in the database engine uses several hundred megabytes of memory while the file cache is much smaller), or we will keep replacing entries in the file cache. File cache replacements may also need to be handled as copy-on-replacements, which can become a concern. These issues lead us to believe that a closer examination of the subtle interactions between the prefetching engine (that runs at user level) and the Linux file cache is needed, so that we understand the full ramifications of the pros and cons of these issues. Such a detailed exploration is well beyond the scope of this paper.

6 Network Subsystem: Characterization and Possible Optimizations

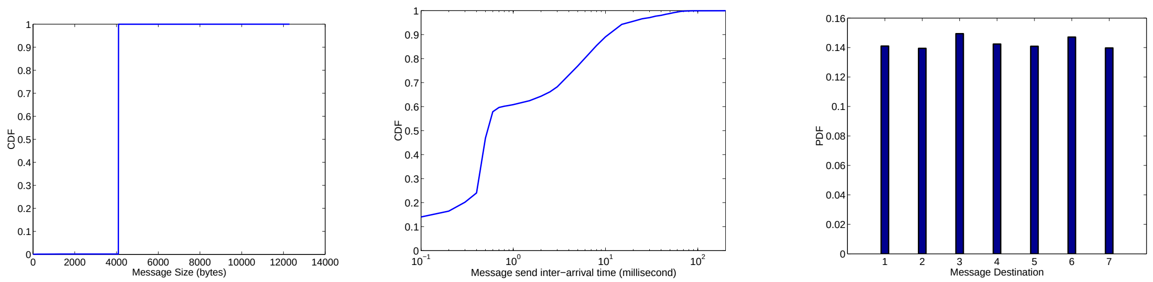
6.1 Characteristics

We next move on to the other exercised system service, namely TCP socket communication. As in the earlier section, we first attempt to characterize this service based on certain metrics that we feel are important for optimization. We examine the message exchanges based on the following characteristics: the message sizes, the inter-injection time (between successive messages by an application), and the destination for a message. We present these characteristics using density functions. The inter-injection time (or injection rate) and message size properties are captured by drawing their corresponding Cumulative Density Functions (CDF). The destination for a message is captured by a Probability Density Function (PDF) showing the probability of a message from a node heading to a specific node (7 possibilities on a 8 node cluster).

In the interest of space, We show the network subsystem characteristics pictorially in Figure 4 for query Q16, which has the highest message injection rates shown in Table 1. Many of the results are similar across queries, and we explicitly mention the differences in the text if there are any. These results have been obtained by instrumenting the kernel and logging all the socket events, their timestamps and arguments at the system call interface. As will be pointed out later on, for some characteristics we also needed to log messages themselves or at least their checksums.

From the density function graphs, we observe the following:

- The message length CDF graph shows that just a handful of message sizes are used by the database engine. In fact, we observed messages were usually either 56 bytes or 4000 bytes. We hypothesize that the shorter size (56 bytes) is used for control messages, and the larger size (4000 bytes) is used for actual data packets. Other message sizes were not very common. We found that Q1, Q6 and Q7 only send short messages, and short messages are the dominant part of Q15 communication (though there are a few long messages here as well). In the rest of the queries, we observed that there were around 40% short messages on the average.
- There are many messages that are sent out in close proximity (temporally). In fact, Figure 4 (b) shows that



(a)CDF of message size (Q16) (b) CDF of inter-arrival time (Q16) (c) PDF of message destination (Q16)

Figure 4: Characterizing Message Sends

nearly 60% of the messages are separated by less than 1 millisecond from each other temporally. In fact, the temporal separations are much lower for queries Q2, Q7, Q10, Q13, Q14, Q15, Q16, Q18 and Q19. We found similar observations for most of the other queries as well.

- The destination PDF graph is considerably influenced by the nature of database operations. In this engine, joins usually involve all-to-all communication of their corresponding portions of the table, and thus queries that are join intensive (such as Q11 and Q16 that are shown here) have the PDF evenly distributed across the nodes. There are a few queries, such as Q7, that are not really join intensive, but perform more specific operations that are based on values of certain primary keys. With such executions, there is a slight bias in communication towards nodes that have those values. Further, the engine uses a coordinator node that manages the query execution across the cluster, and this node performs a little different (we mentioned earlier that communication pattern was one issue that was different across nodes) than the others. In the other nodes, joins dominate most of the queries in general, and the communication load is more or less balanced.

6.2 Recommendations and Possible Optimizations

The above message characteristics say that messages are clustered together, often coming in close temporal proximity. Further, database operations such as joins use all-to-all communication of messages which have a high probability of being the same size. These observations suggest an hypothesis that many of these messages may actually be point-to-point implementations of a multicast/broadcast that the database engine would like to perform. It should be noted that a multicast can send the same information to several nodes at a much lower cost than sending individual point-to-point messages. This saves several overheads at a node (copies, packetization, protocol header compositions, buffer management, etc.) and can also reduce network traffic/congestion if the hardware supported it. Possible implementations of multicast are discussed later in this discussion.

To find out how many of the message exchanges can be modeled as multicasts, we investigated several approaches. During the execution, in addition to the above events, we also logged the messages themselves. These logs were then subsequently processed to compare whether successive messages were identical and addressed to different destinations. Another approach that we tried (which is actually a possible one to use within the OS during the course of execution itself to detect multicasts) is to compare checksums of successive messages. We would like to point out that we found that messages do indeed differ in the first 56 bytes, and that too in only 3 of these 56 bytes in most cases. After numerous experiments, we feel that the first 56 bytes are the header/control information, and the 3 bytes that really vary include the destination id and a possible sequence number. In our analysis of tracking the number of multicast possibilities, we ignore these three bytes, and check the differences for the rest to verify whether they contain the same information. If they do, then we identify that message as a multicast possibility (and the number of messages that would be incurred in a system that supports multicast would go down in this case). We found that both the approaches - actually comparing the messages or comparing the checksums - gave us similar results, and Table 5 gives the percentage of reduction in the number of messages that would be sent if the underlying infrastructure supported multicasts. This information is given for both the short and long messages to verify if multicasts are beneficial to any one class of messages or for both.

We find that there is a substantial multicast potential in these queries. There is a reduction in the total number of messages ranging from 8% to as high as 76%. In general, we find that the multicast potential is greater for the smaller (possibly control) messages than the longer (possibly data) messages. As was pointed out in the earlier results, both short and long messages are equally common in many queries, and we need to optimize both these classes.

This potential can be realized only if the underlying network supports multicasts (incidentally, we found a large number of these multicasts are in fact broadcasts, which Ethernet can support). Even assuming that the underlying infrastructure (either at the network interface level, or in the physical network implementation) supports multicast, the message ex-

query	% reduction of total messages	% reduction of small messages	% reduction of large messages	query	% reduction of total messages	% reduction of small messages	% reduction of large messages
Q1	44.7	71.4	38.7	Q11	9.6	28.6	0.1
Q2	20.4	58.7	0.2	Q12	8.3	7.8	2.9
Q3	48.2	64.3	38.0	Q13	24.5	75.2	0.1
Q4	22.6	58.6	0.1	Q14	27.9	80.4	0.7
Q5	8.0	7.1	8.4	Q15	46.6	56.5	0.7
Q6	76.4	78.6	45.5	Q16	59.1	63.0	56.9
Q7	57.5	71.4	56.2	Q17	41.5	66.7	27.3
Q8	29.1	75.5	4.8	Q18	11.4	32.3	0.00
Q9	66.8	78.5	61.1	Q19	26.7	79.4	0.2
Q10	25.0	73.6	0.1	Q20	21.1	62.8	0.1

Table 5: The potential impact of multicast on queries

changes should be injected into this infrastructure as multicast messages. This can be done at two levels. First, the application (i.e. the database engine) can itself inject multicast messages into the system. Our conversations with DB2 developers indicate that multicast is used by this database engine for purposes like replication, fault recovery etc., but not extensively for data exchanges when processing a query. Further, to work across numerous different platforms, sometimes it may be easier from the programming viewpoint for these applications to simply treat multicasts as point-to-point messages. The other approach, which we investigate, is to automatically detect multicast messages within the operating system (or middleware before going to sockets) and perform the optimizations accordingly. We next describe an online mechanism for such automatic detection. It should be noted that Table 5 gives an upper bound on message reductions by an offline analysis of the message traces, and the online version has only limited window of events to examine for detecting multicasts.

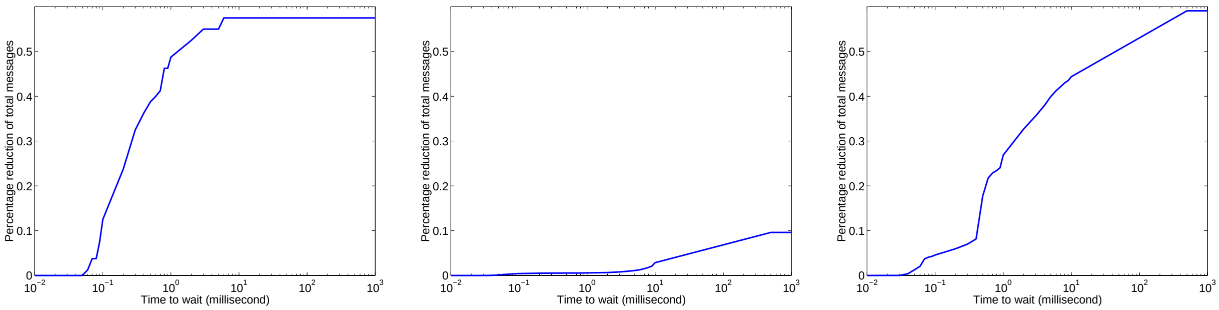
The online algorithm in the OS or middleware can make the system wait for a certain time window while collecting messages detected as multicasts, without actually sending these out. At the end of this window, we send a single multicast message for all the corresponding destinations of the saved messages. The advantage with this approach is that we do not send a message to a destination that the application does not send to. The drawback is that the time between successive messages and time window may be too long a wait that it may be better off just sending them as point-to-point messages. Further, if the window is not long enough, we may not detect some multicasts, and end up sending point-to-point messages.

Consequently, it is important to understand the impact of window size on multicast potential with this online algorithm. If we use such an algorithm within the OS/middleware, then the percentage reduction in the number of messages that need to be sent out with this approach is given in Figure 5 as a function of the time window that it waits for Q7, Q11, and Q16. As is to be expected, expanding the window captures a large fraction of multicasts until the benefits taper off. However, one cannot keep expanding a window arbitrarily since this can slow down the application's forward progress in case this message is needed immediately at the destination. We noticed that the TCP socket implementation on the underlying platform had one-way end-to-end latencies of around 100 microseconds. So it is not unreasonable to wait for comparable time windows since the message would anyway take a large fraction of that time to leave that node. If we consider, window wait times of say 500 microseconds, then we can see reduction of around 40% and 25% of the messages for Q7 and Q16 respectively. On the other hand, Q11 does not benefit much from such an online algorithm (nor from the offline algorithm). We found that queries Q6, Q7, Q9, Q15, Q16, Q17 (the graphs are not explicitly given here) had at least 15% message reduction with a wait time of 500 microseconds. It is, however, important to understand the ramifications of this wait time on query execution for eventual savings. Another point to note is that, if the send on one node and corresponding receive on another are not closely tied to each other (i.e. there is some temporal slackness), then one can increase the time window more aggressively. The underlying protocol can perhaps be extended to carry this slackness information back and forth to enable such decisions.

The observation about online monitoring of slackness, suggests that a more realistic implementation of this algorithm should make adaptive changes to the time window during the course of execution. As it finds that despite waiting, it is not able to combine messages as multicasts, it can adaptively decrease the window so that query execution does not get slowed down (in fact, it can do this even when it finds all the possible multicasts within a shorter time). Similarly, when the underlying protocol detects more slackness (this can be done at the receiver by examining the time difference between when the message gets in and when it is actually used), the algorithm can adaptively increase the window. The window may also need to be tuned based on the message size.

7 Optimizing I/O and Communication Simultaneously

One other issue that we considered for optimizing I/O and communication at the same time was the reduction of copies when there is the possibility of reading from disk and simply sending the data out to another cluster node. A system with a storage area network or a network attached storage disk (NASD) would facilitate direct access of remote data by a node, but the environment that we are conducting the eval-



(a) Q7 (b) Q11 (c) Q16

Figure 5: The impact of wait time on multicast message detection

uations on do not provide such capabilities. A node has to necessarily involve the remote CPU (specifically its counterpart database process) to retrieve data from its disk. Similarly the CPU has to be involved in the write to remote disk as well. Some of the optimizations that an OS could do, when there is no direct remote access hardware support, is to optimize the overheads that are involved when moving data from one I/O channel (say the disk) to another, is to manage buffers effectively by reducing copying costs [12, 5].

To evaluate the possibilities with this approach, we instrumented all the socket and I/O calls, and compared all the data that is read/written from disks and the socket messages. However, we did not find very close similarity in the content across these channels to suggest significant benefits from this approach. This can be attributed to the fact that several queries require a node to send out specific columns of a table after reading from disk. This usually requires retrieving all the columns of a row from the disk since it is stored in row-major order on disk, and then selecting the columns to send out. Sometimes, only select rows need to be sent based on some predicate. Such filtering/selection operations cannot be offloaded to Linux (to ask it to read from disk and perform these operations and then send the result out on sockets, thus reducing crossing protection boundaries and copying). Hence, we do not see too much scope for optimizing these mechanism unless the OS can allow extensibility/modularity to perform such operations. Earlier studies examining cross-channel buffer management [12] showed benefits for web servers, where the data is not really processed/filtered as is the case for these database engines. Further, in a cluster with hardware capabilities like a storage area network or NASD, it would be useful to incorporate intelligence at the disk so that these operations can be carried out to reduce transfer traffic as pointed out by others [11, 13, 15].

8 Summary of Results and Concluding Remarks

This is the first study to embark on a detailed characterization and to present a range of performance statistics for the execution of TPC-H queries on a medium sized Linux cluster of SMP nodes (a popular configuration in today's commercial market) connected by Myrinet and Ethernet. This has required distributing the tables of this workload across the disks of the cluster, implementing the queries, detailed kernel instrumentation to log events, and kernel modifications/extensions to better understand the interaction of the database server with the OS. A brief summary of the issues that are observed from the evaluation follows.

Moving from a uniprocessor/SMP to a cluster does not make I/O any less important for a database engine. We find that disk activity can push CPU utilization as low as 30% in some queries. The overhead of I/O is not just because of the disk latencies, and a significant portion is in the pread system call itself (copying costs mainly). Further, most of the I/O activity is because of the database prefetcher, that brings in large chunks of data (32 blocks at a time), ahead of use, and this is able to do a fairly good job because of reasonably good regularity/sequentiality in the queries. The read ahead feature of the Linux file cache, further helps in reducing the overheads providing hit rates of around 50%. While prefetching mechanisms, whether in the database engine or in the Linux file cache, can be tailored (it probably already is) for such sequentiality, the only consideration is the buffer space availability which in turn depends on physical memory availability. On the other hand, we find that it is extremely important to optimize the pread system call itself, by reducing the amount of copying. In this decision support workload that is read dominated, reducing copies can significantly reduce read overheads without sacrificing much on sharing costs. One could afford to pay higher penalties at writes if needed, if that can cut down read costs significantly. There are several known techniques for reducing copying costs , and in this study we examined the virtual address remapping scheme to show how many copies can actually be avoided. This scheme helps us achieve the objective without requiring any modifications to the legacy database code, but a more detailed investigation of the cost of address remappings is warranted since this can become expensive as pointed out earlier . It is our belief that asynchronous I/O provisioning in Linux [1] would also help, though this would again require application modifications.

The other system service that is also exercised is the socket communication to exchange control and data messages amongst the nodes. While this may not be as dominant as I/O, we find 5-10% of the execution time is spent in socket calls even for a 8-node cluster, and this issue will become more important for larger clusters. We find that many of these messages are identical, suggesting potential for multicasts/broadcasts, which the database engine implements as point-to-point messages.

It should be noted that our goal in this paper is not to recommend specific implementations or designs for improving performance. Rather, we are trying to identify characteristics of application-OS interactions and to suggest issues that can help improve performance for this workload.

Our ongoing work is examining how best to provide the support that the database engine requires, and how to manage the resources effectively in the presence of multiple users. In addition, we are investigating other TPC workloads, as well as other cluster applications such as web and multimedia services for similar studies.

References

[1] POSIX Asynchronous I/O. `http://oss.sgi.com/projects/kaio`.

[2] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proc. of the 15th ACM. Symp. on Operating Systems Principles*, December 1995.

[3] J. Catozzi and S. Rabinovici. Operating System Extensions for the Teradata Parallel VLDB. In *Proceedings of Very Large Databases Conference*, pages 679–682, 2001.

[4] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: a Hardware/Software Approach*. Morgan Kaufmann Publishers, San Francisco, CA, 1999.

[5] P. Druschel and L. L. Peterson. Fbufs: A highbandwidth cross-domain transfer facility. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pages 189–202, 1993.

[6] D. R. Engler, M. Frans Kaashoek, and J. W. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proc. of the 15th ACM. Symp. on Operating Systems Principles*, December 1995.

[7] W. W. Hsu, A. J. Smith, and H. C. Young. Analysis of the Characteristics of Production Database Workloads and Comparison with the TPC Benchmarks. *IBM Systems Journal*, 40(3), 2001.

[8] W. W. Hsu, A. J. Smith, and H. C. Young. I/O Reference Behavior of Production Database Workloads and the TPC Benchmarks - An Analysis at the Logical Level. *To appear in ACM Transactions on Database Systems*, 2001.

[9] M. A. Kandaswamy and R. L. Knighten. I/O Phase Characterization of TPC-H Query Operations. In *Proceedings of the 4th International Computer Performance and Dependability Symposium*, March 2000.

[10] K. Keeton. *Computer Architecture Support for Database Applications*. PhD thesis, Dept. of Computer Science, The University of California at Berkeley, Fall 1999.

[11] K. Keeton, D. A. Patterson, and J. M. Hellerstein. A Case for Intelligent Disks (IDISKs). *SIGMOD Record*, 27(3):42–52, September 1998.

[12] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. *ACM Transactions on Computer Systems*, 18(1), 2000.

[13] E. Riedel, C. Faloutsos, G. A. Gibson, and D. Nagle. Active Disks for Large-Scale Data Processing. *IEEE computer*, 34(6):68–74, June 2001.

[14] M. N. Thadani and Y. A. Khalidi. An efficient zero-copy I/O framework for UNIX. Technical Report SMLI TR-95-39, Sun Microsystems Laboratories, Inc., May 1995.

[15] M. Uysal, A. Acharya, and J. Saltz. Evaluation of Active Disks for Decision Support Databases. In *Proceedings of the Sixth International Symposium on High Performance Computer Architecture*, 2000.

[16] S. Venkatarman. Global memory management for multi-server database systems. Technical Report CS-TR-1996-1325, Univ. of Wisconsin, 1996.