A Performance Evaluation of Parallel Information Retrieval on Symmetrical Multiprocessors

Zhihong Lu Kathryn S. McKinley Brendon Cahoon

Department of Computer Science
University of Massachusetts
Amherst, MA 01003
{zlu, mckinley, cahoon}@cs.umass.edu

Abstract

Providing timely access to text collections both locally and across the Internet is instrumental in making information retrieval (IR) truly useful. In this paper, we investigate how to exploit a symmetrical multiprocessor architecture to build high performance IR servers. We start by implementing a multithreaded IR server and a multitasking IR server to investigate that how best to execute multiple IR commands in parallel. We use InQuery, an inference network, full-text IR retrieval engine, to provide the basic IR services [5, 6, 20]. To expedite our investigation of possible system configurations, characteristics of IR collections, and the basic IR system performance, we implement a simulator with numerous system parameters, such as the number of CPUs, threads, disks, collection size, and query characteristics. We then validate the simulator against our implementation. By using multiple threads, CPUs, and disks, we demonstrate scalable performance for a variety of system configurations. We also find bottlenecks where additional threads, CPUs, and disks either degrade or have no impact on performance. Our results suggest that information retrieval is easily parallelized, but because it performs significant amounts of both I/O and CPU processing, to produce scalable performance requires a careful balance of hardware resources.

1 Introduction

As information explodes across the Internet and elsewhere, providing fast and effective information retrieval is becoming increasingly important. In this paper, we investigate how to exploit a symmetrical multiprocessor (SMP) architecture to achieve high performance for information retrieval servers. Information retrieval (IR) is an ideal application to parallelize. Queries and other IR commands are independent. IR systems can easily divide collections across multiple disks, search the resulting sub-collections independently, and then merge the results. However, because the IR workload is heterogeneous, i.e., it consists of significant amounts of both I/O and CPU processing, simply adding more disks or CPUs does not necessarily produce scalable performance. We investigate how best to execute multiple IR commands in parallel using multithreading, multitasking, and multiple disks on a SMP. We explore a wide range of system parameters to balance CPU and I/O across our application. We also investigate the performance effects of partitioning a single collection across multiple disks. We find bottlenecks and in many instances, show scalable performance for small numbers of processors.

Our IR server is based on InQuery, a full-text inference network based information retrieval engine [5, 6, 20]. Our work is novel because it investigates a real, proven effective system under a variety of realistic workloads and hardware configurations on SMP architecture. The previous work either investigates the IR system on massively parallel processing(MPP) architecture [1, 9, 11, 15, 17, 18, 19] or it investigates only a subset of the system on SMP architecture such as the disk system [14] or it compare the cost factors of SMP architecture with other architectures [8]. (Section 4 compares our work to previous research on parallel information retrieval in more detail.)

We started by building a parallel IR server for an SMP architecture, where all CPUs, disks, and memory are shared and communicate on a shared bus (see Figure 1). We implemented both a multitasking version, and a multithreaded version. We also use a simulator to expedite our investigation of possible system configurations, characteristics of the IR collections, and the basic IR system performance. For example, our

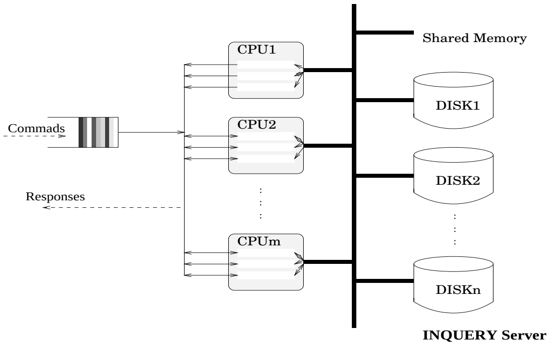


Figure 1: The parallel InQuery server

simulator can vary the number of CPUs, threads, disks, the collection size, query frequencies, query lengths, and workloads. We validate the simulator with the prototype for several interesting system configurations. We describe our IR system, the simulator and its validation in Section 2.

Section 3 presents results that demonstrate the performance improvements and limitations due to multithreading, multitasking, collection size, adding CPUs, and adding disks. We use a response time of 10 seconds as an upper bound on acceptable response time. We begin by investigating the benefits of multithreading verses multitasking using the IR server implementations. We find their performance is similar, although the multithreaded version is always slightly faster than multitasking (90% of the measured response times fall within 10% of each other). The performance benefit of multithreading is thus enough to warrant its inclusion in any new parallel IR system developed from scratch, but it may not be worth the recoding effort in a large legacy system with pervasive global variables.

In Section 3.2.1, we use the simulator to show that for a basic system with one CPU and one disk, multithreading yields performance improvements until the disk becomes overwhelmed. For example, going from 1 to 4 threads increases the number of requests with response time under 10 seconds from 60 to 90 per minute, a factor of 1.5 improvement. Disk utilization simultaneously increases from 70.8% to 88.6% which prevents additional improvements with more than 4 threads.

To relieve the disk bottleneck, we add disks, and then add CPUs as they become the bottleneck. Our results show that if the CPUs and disks are not balanced, the additional hardware can actually degrade performance. For example, for 1 CPU with 16 threads, using the round-robin strategy to partition a 1 GB collection over 4 disks performs worse than over 2 disks, because the CPU cannot keep up with the disks. Increasing the number of CPUs from 2 to 4 for a 1 GB collection on one disk does not improve performance either, but in this case it is because the disks are the bottleneck. The right balance for our system seems to be 2 disks with 1 GB or less data a piece per CPU with 4 threads.

In Section 3.3, we examine system scalability as the collection size increases from 1 GB to 16 GB. We show in several cases that our system can search more data with no loss in performance. For example, using a round-robin collection partition over 16 disks and enables us to increase the collection size from 4 GB to 8 GB with virtually no impact on performance for 4 CPUs with 16 threads and an arrival rate of up to 90 requests per minute. Although performance eventually degrades as the collection size increases, we demonstrate system configurations for which the performance degrades very gracefully.

The remainder of this paper is organized as follows. The next section describes our implementation, simulator, and the validation. Section 3 presents experiments that measure the performance and scalability of our system. Section 4 compares this work to previous work, and Section 5 summarizes our results and concludes.

2 A Parallel Information Retrieval Server

This section describes the implementation of our parallel IR server, and simulator. We begin with a brief description of the InQuery retrieval engine [5, 6, 13], the features we model, and a validation of our simulation of this basic functionality. We then describe the multithreaded and multitasking implementations, and validate our simulator against the multithreaded implementation.

2.1 InQuery Retrieval Engine

2.1.1 InQuery

InQuery is one of the most powerful and advanced full-text information retrieval engines in commercial or government use today [13]. It uses an inference network model, which applies Bayesian inference networks to represent documents and queries, and views information retrieval as an inference or evidential reasoning process[5, 6, 20].

An InQuery database consists of original document files, an inverted file of terms, an inverted file of field-based terms, a stopword dictionary, a file storing processing information, a file of the most frequently occurring terms, and a viewing database which store offsets of documents in the original document files. An inverted file contains term keys, their corresponding lists of documents, and frequency and position information in the original document files. Either a custom B* tree package with concurrency control or the Mneme persistent object store [2] manages the inverted files. The indexing overhead for a collection of documents is 30% to 40% of its original data size. For example, a 1.2 GB Tipster 1 collection [5] needs 0.5 GB extra disk space to store indexes. InQuery accepts both natural language and structured queries.

The InQuery server supports a wide range of IR commands, such as query, document, and relevance feedback. The three basic IR commands are **query**, **summary** and **document** commands. A **query command** requests documents that match a set of terms. A query response consists of a list of top ranked document identifiers. A **summary command** consists of a set of document identifiers. A summary response includes the document titles and the first few sentences of the documents. A **document command** requests a document using its document identifier. The response includes the complete text of the document.

2.1.2 Simulation Model

We use a simulation model we previously built for InQuery work [3, 4]. Because we use a more recent version of InQuery on a DEC AlphaServer 2100 5/250 clocked at 250 MHz instead of an MIPS R3000 clocked at 40 MHz, we validated query response time of our simulator again. We model query response time as a function of query length and term frequency. Our validation demonstrated that all queries fall within 40% of the actual system, and of the queries we do not accurately simulate, we usually over estimate rather than under estimate. We are more accurate on long queries. We describe this validation in more detail in Appendix A.

2.2 The Parallel IR server

In this section, we describe the multithreaded and multitasking IR server implementations. We present the simulator for these systems, and the system parameters we use through out the rest of the paper. We then validate the simulator of the parallel IR server. The parallel IR servers exploits parallelism as follows: (1) It executes multiple IR commands in parallel, by either multitasking or multithreading; or (2) It executes one command against multiple partitions of a collection in parallel.

2.2.1 Multithreading vs. Multitasking

Since we already have distributed servers [3, 4] where each server is single-thread process built on a uniprocessor machine, we implemented a parallel server via multitasking. This version simply executes a light-weight broker and multiple executables of the single-thread server on the same machine, communicating by message passing [3, 4].

Another natural implementation of the parallel InQuery server is to use a thread package to build a shared-everything version, i.e., a multithreaded version. We use the POSIX thread package [16]. Because threads within a process share the same virtual address space, context switching between threads is less expensive than that between processes. In addition, the cooperating threads communicate by simply accessing synchronized global or static variables; thus, we expect the multithreading to be more efficient than the multitasking. Because multithreaded programming has only recently become common, many existing programs are not easy to thread due to pervasive global and static variables. All previous versions of InQuery have this problem, and we spent almost a month eliminating these variables from a subset of the InQuery system.

Parameters	Abbreviation	Values
Query Number	QN	50 1000
Terms per Query (average) shifted neg. binomial dist. (n,p,s)	TPQ	2 (4,0.8,1)
Query Term Frequency dist. from queries	QTF	Observed Distribution
Client Arrival Pattern Poisson process (requests per minute)	λ	6 30 60 90 120 150 180 240 300
Collection Size (GB)	CSIZE	1 2 4 8 16
Disk Number	DISK	1 2 4 8 16
CPU Number	CPU	1 2 3 4
Thread Number	TH	1 2 4 8 16 32

Table 1: Experimental Parameters

We compare the performance of the multithreaded version and the multitasking version in Section 3.1.

2.2.2 Simulator

This section describes the additional features and their parameters that we use to model the parallel IR server, and its validation. To model a parallel IR server, we extended the simulator to model threads, multiple CPUs, and multiple disks. The system parameters also include the original parameters: query length, query term frequency, client arrival rate, and collection size. Table 1 presents all the parameters and the values we use in our experiments and validation.

We assume the client arrival rate as a Poisson process. Each client issues a query and waits for response. For each query, the server performs two operations: query evaluation and retrieving the corresponding summaries. Since users typically enter short queries [10], we experiment with a query set that consists of 1000 short queries, with an average length of 2 that mimic those found in the 103rd Congressional Record query sets [10], and use an *observed* query term frequency distribution obtained from their distribution in the Tipster query sets [5].

We vary the arrival rate and the size of collection in order to examine the scalability of the server as the number of clients and the size of the collection increases. We also vary the number of CPUs, disks, and threads in order to investigate the effects of changing system configurations. All experiments measure

Query Type	Arrival Rate λ (per min.)	Number of Threads					
		1	2	3	4	8	16
Short	6	2.2%	3.1%	-2.8%	2.8%	3.0%	2.2%
	30	1.5%	0.5%	-2.5%	2.5%	-0.8%	1.5%
	60	13.7%	3.0%	1.9%	-1.5%	1.0%	-0.3%
	300	27.2%	17.6%	3.8%	-0.4%	2.1%	1.0%
Medium	6	5.7%	3.9%	6.0%	-1.1%	-0.5%	1.3%
	30	19.8%	9.0%	4.4%	1.9%	1.2%	0.5%
	60	26.0%	12.0%	2.5%	-3.9%	1.9%	-0.4%
	300	21.5%	15.7%	7.2%	8.8%	5.0%	4.1%
Long	6	1.3%	8.0%	0.8%	1.0%	1.2%	0.1%
	30	15.8%	6.6%	-3.8%	-0.4%	0.4%	-0.5%
	60	10.1%	1.8%	-2.1%	1.8%	1.0%	1.7%
	300	12.7%	10.3%	1.7%	2.4%	3.9%	3.1%
All		13.1%	7.6%	2.3%	1.2%	1.6%	1.2%

Table 2: Percentage Difference of Average Response Times between the Implementation and Simulator

response time, and CPU and disk utilization, and determine the largest arrival rate at which the system supports a response time under 10 seconds.

Validation of Query Operation

This section validates query operations on a multithreaded server with a configuration of a single CPU and disk as the query arrival rate and the number of threads increase. Each thread executes one query. Since we assume that queries arrive as a Poisson process, we assume the inverted file is in memory. We call this a warm start. A cold start is when we assume the inverted file is not initially in memory.

Table 2 lists the percentage difference of average response time between the actual system and the simulator for each query set as the arrival rate and the number of threads increase. Positive numbers indicate the simulator overestimates the actual system. On average, the simulator is 4.5% slower than the actual system. The difference between the actual system and the simulator tends to decrease as the number of threads increases and the query arrival rate decreases. The difference between the simulator and the actual system increases as a function of the number of queries waiting in the queue. Overall, the simulator matches the actual system closely.

Arrival	Number of Threads or Processes								
Rate	1			2			3		
(per min.)				thr	proc	diff	thr	proc	diff
6	2.78			2.20	2.21	-0.4%	2.10	2.30	-9.0%
30	8.12			3.72	4.02	-7.2%	3.13	3.43	-9.0%
60	46.50			20.14	22.08	-9.6%	18.49	21.62	-16.9%
300	72.50			52.85	59.24	-12.0%	46.80	49.97	-6.7%

Arrival	Number of Threads or Processes								
Rate	6			9			12		
(per min.)	thr	proc	diff	thr	proc	diff	thr	proc	diff
6	2.20	2.37	-7.7%	2.33	2.37	-1.7%	2.35	2.36	-0.4%
30	3.50	3.72	-6.2%	3.62	3.68	-1.6%	3.58	4.05	-13.1%
60	17.45	17.72	-1.5%	15.60	16.10	-3.2%	16.19	16.20	-0.6%
300	38.73	40.16	-3.6%	40.99	41.32	-0.8%	41.67	45.32	-8.8%

Table 3: Actual Measured Average Response Time on a Cold Start (seconds)

3 Experiments and Results

This section explores how best to use multithreading, multitasking, and collection partitioning on a SMP with multiple disks to improve the performance of a parallel IR server. We use our implementations to investigate the benefits of the multithreading versus multitasking. We use the simulator to investigate system performance and scalability under a variety of workloads and hardware configurations.

3.1 Multithreading vs. Multitasking

This section compares the performance of the multithreaded version and the multitasking version of the IR server on a Alpha Server 2100 5/250 with 1 GB of memory and 3 CPUs. We use 1.2 GB Tipster 1 text collection built as a single database and on a single disk. The inverted file (.5 GB) thus fits in memory (1 GB). The query set is 50 queries generated from the description fields of TIPSTER topics 51-100. Each query is simply a sum of the terms in the corresponding description field, with an average of 8 terms per query. We simulate the query arrival as a Poisson process. In the multithreaded version, the server starts a set of threads and then assigns each query to a single thread. In the multitasking version, the server starts a set of processes and then assigns each query to a single process. We measure the average response time of query evaluation and the corresponding summary response information for the relevant documents.

In Table 3, we measure response time with a cold start such that all term and document accesses cause I/O operations. If a term or document occurs more than once in the queries, only the first access involves

Arrival	Number of Threads or Processes															
Rate	1	2			3			6			9			12		
(per min.)		thr	proc	diff	thr	proc	diff	thr	proc	diff	thr	proc	diff	thr	proc	diff
6	1.23	1.15	1.16	-0.8%	1.16	1.16	0.0%	1.16	1.15	0.8%	1.15	1.16	-0.8%	1.15	1.16	-0.8%
30	1.68	1.20	1.30	-8.3%	1.17	1.26	-7.6%	1.15	1.24	-6.1%	1.15	1.20	-4.3%	1.16	1.21	-4.1%
60	2.83	1.46	1.51	-3.4%	1.26	1.40	-11.1%	1.23	1.30	-5.6%	1.22	1.29	-5.7%	1.23	1.29	-4.8%
300	27.74	12.43	13.16	-7.6%	7.45	7.93	-6.4%	6.59	6.77	-2.7%	6.75	6.76	-0.2%	6.77	6.80	-0.4%

Table 4: Actual Measured Average Response Time on a Warm Start (seconds)

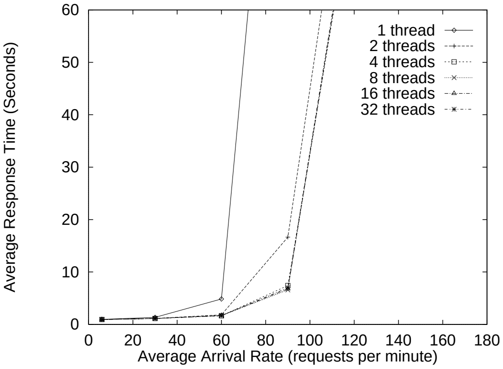
the I/O, since the memory is large enough to cache the inverted file for a 1 GB collection. (1 GB of memory, for a .4 GB inverted file.) Table 4 demonstrates a warm system in which we assume the inverted file is in memory during query and summary evaluation. The measured response times in Tables 3 and 4 are an average of three runs.

In both cases, the multithreaded version is slightly faster than the multitasking version. The differences range from 0.4% to 16.9% depending on the the client arrival rate and the number of threads and processes, with 90% of the measured response time falling within 10% of each other. An experiment with a query set with an average of 27 terms per query shows the same trend.

Both multitasking and multithreading require more memory than a single instance of the system. In InQuery version 3.1, a query with 2 terms assigned to a thread needs roughly 3 MB of additional memory. For example with 16 threads, the system needs at least 48 MB of additional memory. We assume there is enough memory in our experiments.

The multithreaded version is faster than the multitasking version, which suggests we should use multithreading. However, multithreading a large legacy system with pervasive global variables is a non trivial task. To implement multitasking requires the addition of a coordinator process and message passing between the coordinator process and InQuery servers. Multitasking may be preferable, since it is only slightly slower

QN	TPQ	QTF	CSIZE	CPU	DISK	TH
1000	2	Obs.	1 GB	1	1	Varied



Utilization for $\lambda = 90$						
	Number of Threads					
Resources	1	2	4	8	16	32
CPU	26.3%	30.6%	32.6%	32.8%	32.9%	33.1%
DISK	70.8%	81.3%	88.1%	88.6%	88.8%	88.9%

Figure 2: Performance as the number of threads increases (Simulated)

and requires significantly less programming. We implement both versions in their simplest form where only one thread or process is used to evaluate each command. Increasing the granularity of parallelism by partitioning a query or collection may increase the performance advantage of multithreading, since it has less communication overhead than multitasking.

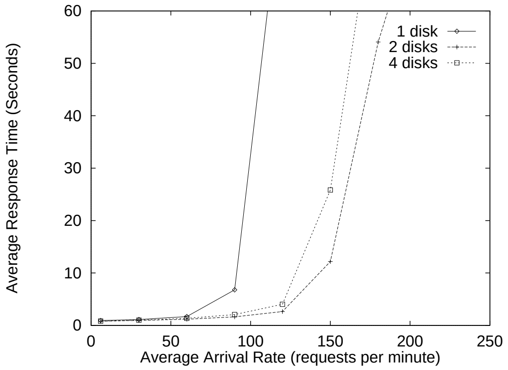
3.2 System Scalability

In this section, we use the simulator to explore system scalability with respect to multiple threads, CPUs, and disks for a 1 GB collection. We assume clients arrive as a Poisson process. For each client, the server performs two operations: query evaluation and retrieving the corresponding summaries. We start with a base system that consists of one thread, CPU, and disk.

3.2.1 Threading

Figure 2 illustrates how the average response time and the resource utilization change as the number of threads increases using 1 CPU for a 1 GB collection on 1 disk. The average response time improves significantly as the number of threads increases from 1 to 4. For example, at an arrival rate of 60 requests per minute, the average response time improves by a factor of 3. The largest arrival rate at which the system

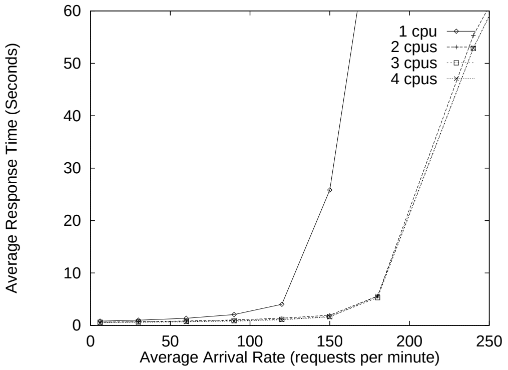
QN	TPQ	QTF	CSIZE	CPU	DISK	TH
1000	2	Obs.	1 GB	1	Varied	16



Utilization for $\lambda = 150$			
	Number of Disks		
Resources	1	2	4
CPU	36.1%	70.4%	91.4%
DISK	97.5%	93.0%	63.6%

Figure 3: Performance as the number of disks increases (Simulated)

QN	TPQ	QTF	CSIZE	CPU	DISK	TH
1000	2	Obs.	1 GB	Varied	4	16



Utilization for $\lambda = 180$				
	Number of CPUs			
Resources	1	2	3	4
CPU	91.6%	56.7%	38.0%	30.5%
DISK	65.0%	79.3%	79.3%	79.7%

Figure 4: Performance as the number of CPUs increases (Simulated)

supports a response time under 10 seconds increases from 60 to 90 requests per minute, a factor of 1.5 improvement.

Performance improves because threading overlaps the CPU and I/O processing, which increases resource utilization (the classic advantage of multiprogramming). The table in Figure 2 presents the CPU and disk utilization for an arrival rate of $\lambda = 90$ requests per minute. The disk becomes saturated when number of threads reaches 4 and before the CPU is saturated. More threads do not yield further performance improvements.

3.2.2 Adding Disks

To eliminate the disk bottleneck, we distribute a collection over several disks to introduce parallel disk accesses, and thus improve disk access time since there is less to search on each disk. When searching a partitioned collection in parallel, we achieve the best performance by *evenly* distributing the workload over all sub-collections. We adopt a round-robin strategy to partition a collection and build a database for each partition. All database components for each partition resides on one disk.

Figure 3 illustrates the average response time and resource utilization as the number of disks increases using one CPU and a round-robin strategy to partition a 1 GB collection. The average response time improves significantly when we partition the collection over two disks. Increasing the arrival rate up to 90 requests per minute has almost no impact on average response time. The largest arrival rate at which the server supports a response time under 10 seconds increases from 90 to 150 requests per minute, a factor of 1.46. Performance degrades however when we partition the collection over four disks.

By examining the utilization of the CPU and disks in the table in Figure 3 for $\lambda = 150$, we see that the computation changes from the I/O to CPU bound as the number of disks increases. The CPU utilization increases from 36.1% to 91.4%. As the number of disks increases, there is additional CPU overhead to access each disk and combine responses. For example, the overhead is doubled for queries when we distribute a collection over two disks. Partitioning over two disks improves performance because querying smaller collections is quicker although not twice as fast, and the searches are in parallel. For summary commands, we gain additional parallelism because the retrieving summaries is typically twice as fast, since the requested documents are usually split between collections and there is minimal overhead. For four disks, the overhead overwhelms the CPU and negates the benefits of the additional parallelism.

3.2.3 Adding CPUs

Figure 4 illustrates the average response time and the resource utilization as the number of CPUs increases when we partition the collection over 4 disks with 16 threads. Recall that for 1 CPU, partitioning over 4 disks causes the CPU to be a bottleneck. The average response time improves significantly, when we add one CPU. At an arrival rate of 90 requests per minute, the average response time improves by a factor of 2; at a arrival rate of 120 requests per minute, the average response time improves by a factor of 2.9. The largest arrival rate at which the server supports a response time under 10 seconds increases by a factor of 1.46, from 130 requests per minute to 190. When $\lambda = 180$ and the number of CPUs varies from 2 to 4, utilization of CPU drops with the number of CPUs, and disk utilization is almost fixed at 79%. Once we relieve the CPU bottleneck going from 1 to 2 CPUs, adding CPUs does not bring significant improvements.

3.2.4 Summary

We improve the performance of our IR server through better software: multithreading; and with additional hardware: CPUs and disks. In our base system with a single thread, disk, and CPU, the server is I/O bound. Since the threads are independent in this system, threading improves performance of the base system by gaining well known multiprogramming benefits - increasing the hardware resource utilization because I/O and computation overlap. For short queries on a 1 GB collection, threading improves the arrival rate at which the system supports a response time under 10 seconds from 60 requests per minute to 90. Adding hardware improves performance, because the IR server can exploit the parallelism of the IR commands. Partitioning the collection across multiple disks introduces a finer-grain execution of IR commands and shifts the balance of the computation from I/O to CPU bound. Adding CPUs also improves performance to a point. For example, using 16 threads, 4 disks and 2 CPUs improves the arrival rate at which the system supports a response time under 10 seconds by a factor of 3.2 over 1 thread, CPU, and disk. However if the hardware components are not balanced, additional hardware can degrade performance. With 16 threads and 1 CPU, partitioning a 1 GB collection over 4 disks performs worse than 2 disks, because the CPU is overloaded; with 16 threads and 1 disk, increasing the number of CPUs from 2 to 4 does not improve performance because the disk is a bottleneck.

3.3 Scalability with Increasing Collection Size

This section examines system scalability as the collection size increases from 1 GB to 16 GB. We assume that each disk stores all database components for at most 1 GB of data. We consider two disk configurations. In the first configuration, we fix the number of disks at N and use the round-robin strategy to partition the collection (M GB) over N disks, where each disk stores all database components for M/N GB of data. In the second configuration, the number of disks increases with the size of the collection. We use a round-robin strategy to partition the collection (M GB) over M disks, where each disk stores all database components for 1 GB of data.

3.3.1 Distributing the collection over a fixed number of disks

Figure 5 and Table 6 illustrate the average response time and resource utilization when the collection size varies from 1 GB to 16 GB and each disk stores a database for 1/16 of the collection. Table 5 reports the largest arrival rates under different configurations at which the system supports a response time under 10 seconds.

Partitioning the collection over 16 disks illustrates when the system is CPU bound. Although performance degrades as the collection size increases, the degradation is closely related to the CPU utilization. In the case of 1 CPU where the CPU is over utilized for 1 GB and 60 requests per minute, increasing the collection size from 1 GB to 16 GB decreases the largest arrival rate at which the system supports a response time under 10 seconds by a factor of 10. In the case of 4 CPUs, where CPUs are over utilized for 1 GB and 180 requests per minute, the performance degrades much more gracefully. For example, increasing the collection size from 1 GB to 4 GB has no impact on average response time for an arrival rate of up to 90 requests per minute (see Figure 5(c)). Increasing the collection size from 1 GB to 16 GB only decreases the arrival rate at which the system supports a response time under 10 seconds by a factor of 3. This set of experiments illustrates dramatic improvements due to additional CPUs. For a 1 GB collection, 4 CPUs improves the arrival rate at which the system supports a response time under 10 seconds by a factor of 3 compared with 1 CPU. For a 16 GB collection, 4 CPUs improve the arrival rate by a factor of 10 compared with 1 CPU.

We also find instances where the system supports the same arrival rates under different system configurations. At these data points, we can support the same performance by doubling the number of CPUs when the collection size increases by a factor of 4. For example, the system supports an arrival rate of 60 requests per minutes, when using 1 CPU for 1 GB, using 2 CPUs for 4 GB, and using 4 CPUs for 16 GB; the system

Number of	Size of Collection (size/16 GB per disk)				
CPUs	1 GB	2 GB	4 GB	8 GB	16 GB
1	60	35	30	10	6
2	120	90	60	35	25
4	180	150	120	90	60

Table 5: Requests per minute with a response time under 10 seconds for a collection distributed over 16 disks

QN	TPQ	QTF	CPU	DISK	TH	λ
1000	2	Obs.	Varied	16	16	120

Number of		Size of Collection (size/16 GB per disk)				
CPUs	Resources	1 GB	2 GB	4 GB	8 GB	16 GB
1	CPU	96.6%	97.4%	98.1%	98.6%	99.0%
	DISK	21.6%	18.4%	14.7%	11.9%	9.8%
2	CPU	78.1%	91.8%	94.2%	95.9%	97.0%
	DISK	35.1%	34.9%	28.6%	23.0%	19.4%
4	CPU	38.9%	49.7%	68.0%	88.0%	93.5%
	DISK	34.9%	37.8%	41.1%	42.6%	37.4%

Table 6: Resource utilization for a collection distributed over 16 disks (Simulated)

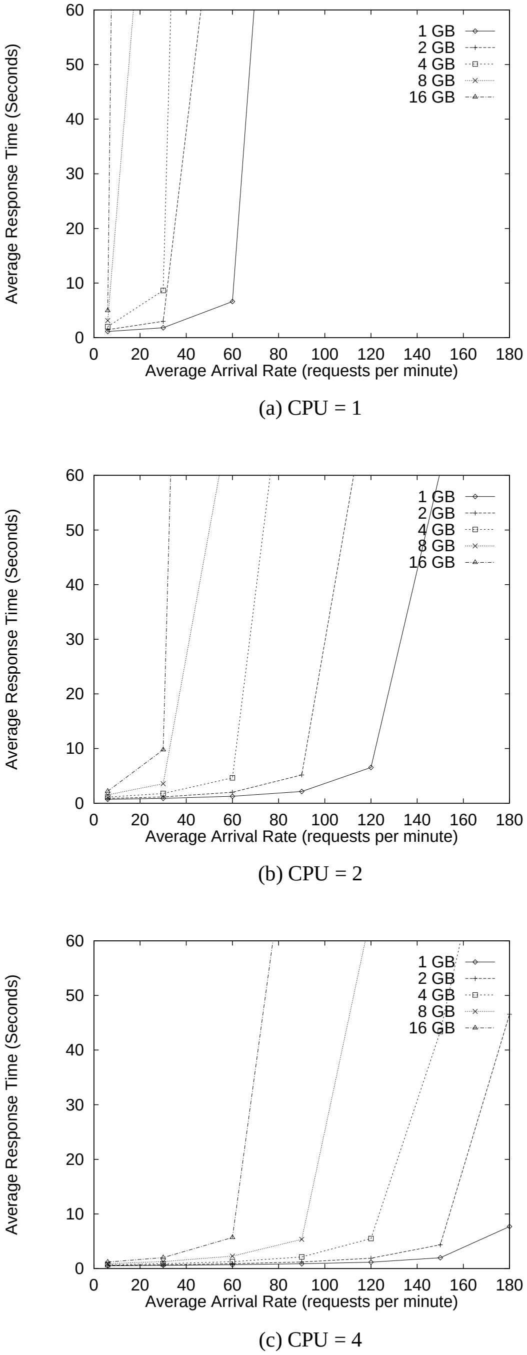
supports the arrival rate of 90 requests per minutes, when using 2 CPUs for 2 GB, and using 4 CPUs for 8 GB.

3.3.2 Distributing the collection over a variable number of disks

Figure 6 and Table 8 illustrate the average response time and the resource utilization when the collection size varies from 1 GB to 16 GB and each disk stores a database for 1 GB of data. Table 7 reports the largest arrival rates under different configurations at which the system supports a response time of 10 seconds.

The results show that the system scales up to 2 GB using 1 CPU, up to 4 GB using 2 CPUs and 8 GB using 4 CPUs. An even more interesting phenomenon is that a single CPU system handles a 2 GB collection faster than a 1 GB collection, and a 4 CPU system handles a 2, 4, or 8 GB collection faster than a 1 GB collection in our configuration. The performance improves because work related to retrieving summaries is distributed over disks such that each disk handles less work, relieving the disk bottleneck. By examining the utilization of CPU and disk in Table 8, we see that the performance improves until the CPUs are over utilized. In the example of the single CPU system, the CPU is over utilized for a 4 GB collection. For a 2

QN	TPQ	QTF	CSIZE	CPU	DISK	TH
1000	2	Obs.	Varied	Varied	16	16



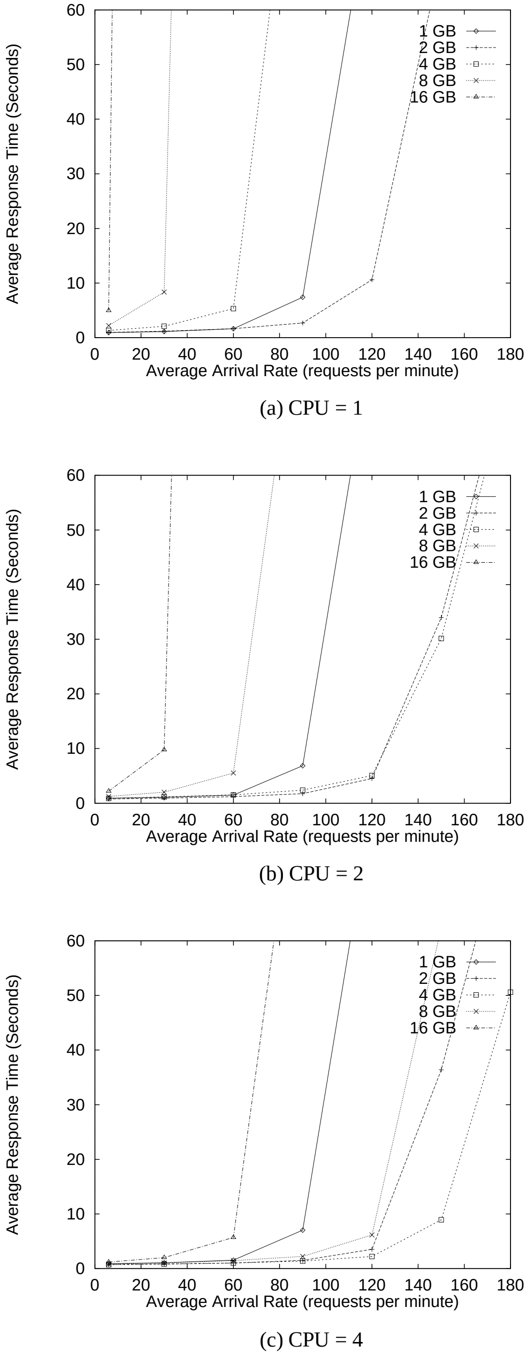
(a) CPU = 1

(b) CPU = 2

(c) CPU = 4

Figure 5: Average response time for a collection distributed over 16 disks (Simulated)

QN	TPQ	QTF	CSIZE	CPU	DISK	TH
1000	2	Obs.	Varied	Varied	Varied	16



(a) CPU = 1

(b) CPU = 2

(c) CPU = 4

Figure 6: Average response time when the number of disks varies with the size of the collection (Simulated)

Number of CPUs	Size of Collection (1 GB per disk)				
	1 GB	2 GB	4 GB	8 GB	16 GB
1	90	115	65	30	6
2	90	125	125	60	30
4	90	125	150	120	60

Table 7: Requests per minute with a response time under 10 seconds when the number of disks varies with the size of the collection

QN	TPQ	QTF	CPU	DISK	TH	λ
1000	2	Obs.	Varied	Varied	16	120

Number of CPUs	Resources	Size of Collection (1 GB per disk)				
		1 GB	2 GB	4 GB	8 GB	16 GB
1	CPU	36.2%	75.3%	94.3%	98.1%	99.0%
	DISK	97.4%	82.2%	43.5%	20.6%	9.8%
2	CPU	18.1%	38.1%	71.2%	95.0%	97.0%
	DISK	97.4%	82.9%	66.2%	40.5%	19.4%
4	CPU	9.0%	19.0%	35.7%	67.5%	93.5%
	DISK	97.4%	82.9%	66.7%	57.1%	37.4%

Table 8: Resource utilization when the number of disks varies with the size of the collection

GB collection distributed over 2 disks, the system handles 27.8% more requests than for a 1 GB collection on 1 disk.

Under this disk configuration, if we want to support the same level of performance, the number of CPUs should be doubled when the amount of data is doubled. For example, if we want the system to support arrival rate of about 60 requests per minute, we need 1 CPU for a 4 GB collection, 2 CPUs for a 8 GB collection, and 4 CPUs for a 16 GB collection.

3.3.3 Summary

In parallelizing our IR server, we find instances when increasing the collection size has no impact in performance. We can even find instances where increasing the collection size improves performance, because each disk handles less work. Although the performance eventually degrades as collection size increases, the degradation is very graceful until the CPU becomes over utilized.

We find that the CPU utilization is more closely related to the number of disks rather than the collection size. For the same amount of data, e.g., 2 GB, partitioning over 2 disk using 2 CPUs results in 38.1% CPU utilization (see Table 8), while partitioning over 16 disks results in 91.8% CPU utilization (see Table 6), due to the addition overhead to access each disk. In this case, the CPU over utilization causes 16 disks to perform worse than 2 disks. However when CPU is not over utilized, the performance improves as we partition the collection over more disks. For example, using 4 CPUs, partitioning 2 GB over 16 disks improves a factor of 1.2 compared with partitioning 2 GB over 2 disks. These results suggest that we need to balance hardware resources carefully in order to achieve scalable performance.

4 Related Work

There have been a number of papers regarding to using multiprocessor machines for information retrieval [1, 8, 9, 11, 14, 15, 17, 18, 19]. Most of them use a distributed memory, massively parallel processing (MPP) architecture [1, 9, 11, 15, 17, 18, 19].

Couvreur *et al.* analyze the tradeoff between performance and cost when searching large text collections. They use simulation models to investigate three different hardware architectures: a mainframe, a collection of RISC processors connected by a network and a special purpose machine [8]. They use different search algorithms on different hardware architectures. The experiments using a mainframe are most related to our work. They measure the response time under different query arrival rates and identify the query arrival rate the system can support within 30-40 seconds. By using a 4-CPU IBM 3090/400E mainframe, they achieve 45 searches per minute when searching a 14 GB collection. In our system, we can achieve 70 searches per minute with a response time under 10 seconds using 4 CPUs when searching a 16 GB collection. Our major contribution is not that we can build a faster system. Since we do not have the figures such as clock speed, memory size of their machines, we cannot compare the numbers directly. Our contribution is that we focus on a single system and analyze how different parameters affect the system performance. Besides measuring response time, we also measure the system utilization and identify bottlenecks.

Jeong and Omiecinski investigate two inverted file partitioning schemes in a shared-everything multiprocessor system [14]. One scheme partitions the posting file by term identifiers while the other scheme partitions the posting file by document identifiers. They focus on the effect of adding disks on system performance. They show that response time decreases as the number of disks increases up to some threshold. Partitioning based on term identifiers performs the best when the term distribution is less skewed or when the term distribution in the query is uniformly distributed. Partitioning based on document identifiers performs the best when term distribution is highly skewed. We address this issue in Section 3.2.2 and Appendix A. Although we only consider one partitioning scheme without much skew in our experiments, we measure the effects of threads, CPUs, and disks.

The other related studies use MPPs and focus on how to speed up single query processing. Stanfill *et al.* implement their IR system on the connection machine (CM), which is a fine-grained, massively parallel distributed-memory SIMD architecture with up to 65,536 processing elements [17, 18, 19]. Bailey and Hwaking report their IR system on Fujitsu AP1000, which is a 128-node distributed-memory multicomputers and each node has a 25 MHZ CPU and 16 MB memory [1]. Cringean *et al.* and Efraimidis *et al.* implement their IR systems on a transputer network, which belongs to the MIMD class of parallel computers [9, 11]. Our work uses a SMP and investigates the system performance when processing multiple queries.

5 Conclusion

In this paper, we investigate building a parallel information retrieval server using a symmetrical multiprocessors to improve the system performance. We measure the actual systems to compare the performance of multithreading and multitasking. We build a flexible simulation model to study performance by varying numerous parameters, such as the number of threads, disks, CPUs and the collection size. We present a series of experiments that measure system response time, utilization, and identify system configurations and workloads at which the system supports a response time under 10 seconds.

The results on multithreading versus multitasking show that the performance of the multithreaded version and the multitasking version is similar in our implementation, although the multithreaded version is always slightly faster than the multitasking version (90% of measured response times fall within 10% of each other). The performance benefit of multithreading is enough to warrant its inclusion in any new parallel IR system developed from scratch, but it may not be worth the recoding effort in a large legacy system with pervasive global variables.

By using the simulator, we first explore system scalability with respect to multiple threads,CPUs, and disks for a 1 GB collection. Our results show that we need threading to increase the hardware resource utilization. We also show that adding hardware components can improve the performance, but these components must be well balanced. In some cases, additional hardware actually degrades performance.

We also investigate the scalability of the server by increasing the collection size from 1 GB to 16 GB. Our results show that we can search more data with no loss in performance in many instances. Although the performance eventually degrades as the collection size increases, the performance degrades very gracefully if we keep the hardware utilization balanced. Our results also show that system performance is more related to the number of disks, rather than the collection size. Our results suggest that system designers for information retrieval should carefully balance their hardware resources in order to achieve good performance.

Acknowledgment

This material is based on work supported in part by the National Science Foundation, Library of Congress and Department of Commerce under cooperative agreement number EEC-9209623. Kathryn S. McKinley is supported by an NSF CAREER award CCR-9624209. Any opinions, findings and conclusions or recommendations expressed in this material are the authors and do not necessarily reflect those of the sponsor.

We would like to thank Bruce Croft, Jamie Callan, and James Allan for their support of this work and all the developers of InQuery without whose efforts this work would not have been possible.

References

[1] P. Bailey and D. Hawking. A parallel architecture for query processing over a terabyte of text. Technical Report TR-CS-96-04, The Australian National University, June 1996.

[2] E.W. Brown, J.P Callan, W.B. Croft, and J.E.B. Moss. Supporting full-text information retrieval with a persistent object store. In *Proceedings of the 4th International Conference on Extending Database Technology (EDBT)*, pages 363–378, Cambridge, UK, March 1994.

[3] B. Cahoon and K. S. McKinley. Performance evaluation of a distributed architecture for information retrieval. In *Proceedings of the Nineteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 110–118, Zurich, Switzerland, August 1996.

[4] B. Cahoon and K. S. McKinley. Evaluating the performance of distributed architectures for information retrieval using a variety of workloads. *Submitted for publication*, 1997.

[5] J. P. Callan, W. B. Croft, and J. Broglio. TREC and TIPSTER experiments with INQUERY. *Information Processing & Management*, 31(3):327–343, May/June 1995.

[6] J. P. Callan, W. B. Croft, and S. M. Harding. The INQUERY retrieval system. In *Proceedings of the 3rd International Conference on Database and Expert System Applications*, Valencia, Spain, September 1992.

[7] J. P. Callan, Z. Lu, and W. B. Croft. Searching distributed collections with inference networks. In *Proceedings of the Eighteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Seattle, WA, July 1995.

[8] T. R. Couvreur, R. N. Benzel, S. F. Miller, D. N. Zeitler, D. L. Lee, M. Singhai, N. Shivaratri, and W. Y. P. Wong. An analysis of performance and cost factors in searching large text databases using parallel search systems. *Journal of the American Society for Information Science*, 7(45):443–464, 1994.

[9] J. K. Cringean, R. England, G. A. Mason, and P. Willett. Parallel text searching in serial files using a processor farm. In *Proceedings of the Thirteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Brussels, Belgium, September 1990.

[10] W. B. Croft, R. Cook, and D. Wilder. Providing government information on the internet: Experiences with THOMAS. In *The Second International Conference on the Theory and Practice of Digital Libraries*, Austin, TX, June 1995.

[11] P. Efraimidis, C. Glymidakis, B. Mamalis, P. Spirakis, and B. Tampakas. Parallel text retrieval on a high performance supercomputer using the vector space model. In *Proceedings of the Eighteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 58–66, Seattle, WA, 1995.

[12] D. Harman, editor. *The First Text REtrieval Conference (TREC-1)*. National Institute of Standards and Technology Special Publication 200-217, Gaithersburg, MD, 1992.

[13] InQuery. An information engine for the u.s. economy. *http://ciir.cs.umass.edu/info/highlights.html*.

[14] B-S. Jeong and E. Omiecinski. Inverted file partitioning schemes in multiple disk systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):142–153, February 1995.

[15] B. Mamalis, O. Spirakis, and Tampakas. Parallel techniques for efficient searching over very large text collections. In *Proceedings of The Fifth Text REtrieval Conference (TREC-5)*, Gaithersburg, MD, 1996. National Institute of Standards and Technology Special Publication.

[16] Open Software Foundation. *OSF DCE Application Development Guide – Core Components*, 1994.

[17] C. Stanfill. Partitioned posting files: A parallel inverted file structure for information retrieval. In *Proceedings of the Thirteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 413–428, Brussels, BELGIUM, 1990.

[18] C. Stanfill and B. Kahle. Parallel free-text search on the connection machine system. *Communications of the ACM*, 29(12):1229–1239, December 1986.

[19] C. Stanfill, R. Thau, and D. Waltz. A parallel indexed algorithm for information retrieval. In *Proceedings of the Twelfth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 88–97, Cambridge, MA, June 1989.

[20] H. R. Turtle. *Inference Networks for Document Retrieval*. PhD thesis, University of Massachusetts, February 1991.

[21] C. L. Viles. *Maintaining retrieval effectiveness in distributed, dynamic information retrieval systems*. PhD thesis, University of Virginia, May 1996.

[22] E. M. Voorhees, N. K. Gupta, and B. Johnson-Laird. Learning collection fusion strategies. In *Proceedings of the Eighteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Seattle, WA, 1995.

Appendix A: Validation of Basic IR Functionality

The simulation model is driven by empirical timing measurements from the actual system. We model three basic IR operations: query evaluation, obtaining summary information, and retrieving documents. We measure CPU and disk usage for each operation, but do not measure the memory and cache effects. We model the collection by obtaining term and document statistics from 1.2 GB Tipster 1 text collection, a standard test collection of full-text articles and abstracts [5, 12]. The simulator only accepts natural language queries. Two parameters, query length and query term frequency, determine the characteristics of a query. (See [3, 4] for more details.) Below, we present new system measurements and their validation.

System Measurements

The system measurements include query evaluation time, and document/summary retrieval time. We obtained the measurements using InQuery version 3.1 running on DEC AlphaServer 2100 5/250 with 3

CPUs (clocked at 250 MHz), 1024 MB main memory and 2007 MB of swap space, running Digital UNIX V3.2D-1 (Rev 41).

We model the query evaluation time as a sum of evaluation time for each term in the query plus a small overhead that represents the time to combine the results of each term [4]. The time to evaluate a term ranges from 0.06 seconds for a term that appears only once in the inverted file to 1.2 seconds for a term that appears 995,008 times (the maximum term frequency of Tipster 1). The evaluation time is divided into CPU and disk access time. The disk access time accounts for 32% to 90% of the total evaluation time with an average of 73%, assuming the index file is on disk.

Since the document sizes of the Tipster 1 collection is not very large (2.3 KB on average) and thus retrieval occurs very quickly, there is no strong correlation between document size and document retrieval time [4]. We thus represent the document retrieval time as a constant value: 0.027 seconds, which is the average document retrieval time for 2000 randomly selected documents from the Tipster 1 collection. The document retrieval time is also divided into CPU and disk access time. The disk access time accounts for 87% of the total retrieval time. We represent the summary retrieval time as a sum of the document retrieval times for each document in the summary request [4].

Validation of Query Evaluation Model

In this section, we validate the accuracy of the query simulation model against the sequential implementation of InQuery version 3.1, by creating artificial queries and comparing the performance of each query on the implementation and simulator. We randomly generate three sets of queries: 50 short queries with an average length of 2, 50 medium queries with an average length of 12, and 50 long queries with an average length of 27. We do not generate queries with multiple occurrences of the same term since our model does not account for these types of queries.

Before processing each query, we chill the system by reading a large file that fills the memory such that every term in the query is read from disk. Table 9 shows the validation results. Column 2 and 3 show

Query	Difference						Ave. Eval.
Type	Ave.	Std.	±10%	±20%	±30%	±40%	Time (sec)
Short	2.0%	16.2%	42%	72%	96%	100%	0.7
Medium	4.2%	10.1%	68%	90%	100%	100%	4.6
Long	2.3%	8.5%	78%	96%	100%	100%	10.1
All	2.8%	11.6%	63%	86%	99%	100%	5.1

Table 9: Query Model Validation

Query	Total		Distribution		
Type	Number	Percentage	0% to -10%	-10% to -20%	-20% to -30%
Short	22	44%	11	7	4
Medium	17	34%	14	3	0
Long	19	38%	18	1	0
All	58	39%	43	11	4

Table 10: Distribution of Underestimated Queries

the average percentage difference of evaluation time between the simulator and the actual system, and its standard deviation. A positive value means that the simulator overestimates the actual system; a negative value means that the simulator underestimates the actual system. Columns 4 thru 7 show the percentage of queries running on the simulator that fall within $\pm 10\%$, $\pm 20\%$, $\pm 30\%$ and $\pm 40\%$ of the actual system. The last column lists the average evaluation query time for each set of queries in the actual system. On average the simulator is 2.8% slower than the actual system with the standard deviation of 11.6%. The variation of short queries is twice that for long queries. All queries fall within 40% of the actual system.

Table 10 details the underestimated queries. Columns 2 and 3 show the total number of underestimated queries and the corresponding percentage in all query sets. Columns 4 thru 6 shows the number of the underestimated queries that fall within -10%, -10% to -20% and -20% to -30% of the actual system. Although we underestimate 58 queries out of 150 queries, 43 queries fall within 10% and only four short queries fall outside of 20%. Thus, we usually overestimate queries. Our validation results show that our query evaluation model matches the actual system very closely, although we do not accurately model every query.