The Hardware/Software Balancing Act for Information Retrieval on Symmetric 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

Web search engines, such as AltaVista and Infoseek, handle tremendous loads by exploiting the parallelism implicit in their tasks and using symmetric multiprocessors to support their services. The web searching problem that they solve is a special case of the more general *information retrieval* (IR) problem of locating documents relevant to the information need of users.

In this paper, we investigate how to exploit a symmetric multiprocessor to build high performance IR servers. Although the problem can be solved by throwing lots of CPU and disk resources at it, the important questions are *how much* of *which* hardware and *what* software structure is needed to effectively exploit hardware resources. We have found, to our surprise, that in some cases adding hardware *degrades* performance rather than improves it. We compare the performance of multithreading and multitasking and show that multiple threads are needed to fully utilize hardware resources. Our investigation is based on InQuery, a state-of-the-art full-text information retrieval engine, that is widely used in Web search engines, large libraries, companies, and government agencies such as Infoseek, Library of Congress, White House, West Publishing, and Lotus.

1 Introduction

As information explodes across the Web and elsewhere, people increasingly depend on search engines to help them to find information. Web searching is a special case of the more general *information retrieval* (IR) problem of locating documents relevant to the information need of users.

Until recently, parallel computers were an expensive and special-purpose tool. Today, most hardware vendors offer affordable symmetric multiprocessors (SMP). Web searching engines, such as AltaVista [1] and Infoseek [15], handle tremendous loads by exploiting the parallelism implicit in their tasks and use SMPs to support their services. Although it is clear that the more CPUs and disks you have the more load the system can handle, the important questions are how much of which hardware and what software structure is needed to exploit these resources. Unfortunately, commercial systems have not published the hardware and software configurations they use to achieve high performance. The previous research investigates either the IR system on massively parallel processing (MPP) architecture [3, 11, 13, 18, 20, 21, 22], or it investigates only a subset of the system on SMP architecture such as the disk system [17] or it compares the cost factors of SMP architecture with other architectures [10].

In this paper, we investigate how to balance hardware and software resources to exploit a symmetric multiprocessor (SMP) architecture to build high performance IR servers. Our IR server is based on InQuery [7, 8, 23], a state-of-the-art full-text information retrieval engine that is widely used in Web search engines, large libraries, companies, and governments such as Infoseek, Library of Congress, White House, West Publishing, and Lotus [16]. Our work is novel because it investigates a real, proven effective system under a variety of realistic workloads and hardware configurations on an SMP architecture with multithreading. Our results provide insights for building high performance IR servers for searching the Web and other environments using a symmetric multiprocessor.

Information retrieval 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 better software: multithreading or multitasking, and additional hardware: multiple disks and CPUs on a SMP. We compare the performance of multithreaded and multitasking implementations of a parallel IR server. We explore a wide range of system parameters to balance CPU and I/O utiliza-

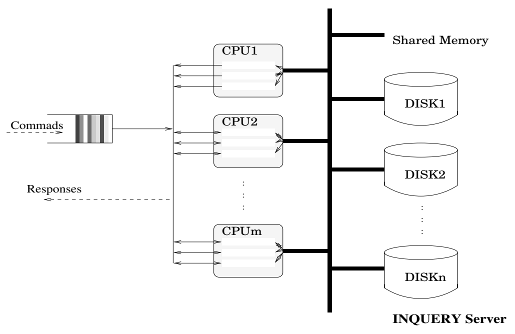


Figure 1: The parallel InQuery server

tion across our application. We also investigate the performance effects of partitioning a single collection across multiple disks. We show scalable performance for small numbers of processors in many cases. We find, to our surprise, that in some cases adding hardware *degrades* performance rather than improves it due to heterogeneous workloads.

2 Overview

To investigate the balance between hardware and software in a IR system, we built 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 and multithreaded version. The server assigns an IR command to one or multiple available processes (or threads), depending on the IR command type and the collection partitioning. 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 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 3.

Section 4 presents results that demonstrate the performance improvements and limitations due to multithreading, multitasking, collection size, adding CPUs, and adding disks. We use 10 seconds arbitrarily as our cutoff for a reasonable response time.[1] 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 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 4.2, we examine system scalability and hardware/software balancing with respect to multiple threads, CPUs, and disks as the collection size increases from 1 GB to 16 GB. We show that the system needs multiple threads to fully utilize hardware resources even for the single CPU and disk configuration. We also demonstrate several configurations in which our system can search increasing amount of data with no loss in performance. Although performance eventually degrades as the collection size increases, we demonstrate system configurations for which the performance degrades very gracefully. If the CPUs and disks are not balanced, we find that the additional hardware can actually degrade performance. Section 5 compares this work to previous work, and Section 6 summarizes our results and concludes.

[1] Commercial systems achieve better response time because they include optimizations such as query result caching that we have not implemented.

3 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 [7, 8, 16], the features we model, and a validation of our simulation of this basic functionality. We also describe the multithreaded and multitasking implementations, and validate our simulator against the multithreaded implementation.

3.1 InQuery Retrieval Engine

3.1.1 InQuery

InQuery is one of the most powerful and advanced full-text information retrieval engines in commercial or government use today [16]. Infoseek, one of most popular Web search engines, is based on the InQuery technology. Large libraries such as the Library of Congress and National Library of Medicine, government agencies such as White House and Internal Revenue Service, and companies such as West Publishing and Lotus also use InQuery to provide different services.

InQuery 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 [7, 8, 23]. In this paper, we use “collection” to refer to a set of documents, and “database” to refer to an indexed collection. 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 [9] package with concurrency control or the Mneme persistent object store [4] 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 [7] needs 0.5 GB extra disk space to store indexes.

The InQuery server supports a wide range of IR commands such as query, document, and relevance feedback. The three basic IR commands we model are **query**, **summary**, and **document** commands. InQuery accepts both natural language and structured queries. 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.

3.1.2 Simulation Model and Validation

We use a simulation model we previously built for InQuery work [5, 6]. 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, I/O bus, 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 well known and used standard test collection distributed by National Institute of Standards and Technology for testing and comparing the current text retrieval techniques [14]. The Tipster 1 collection consists of full-text articles coming from Associate Press Newswire, Wall Street Journal, and Computer Selects (Ziff-Davis Publishing), and abstracts from DOE publications. The average document size of the Tipster 1 collection is 2.3 KB, which is very close to the average Web page size (around 2 KB according to the figures published by AltaVista [2] [2]). The simulator only accepts natural language queries. Two parameters, query length and query term frequency, determine the characteristics of a query. (See [5, 6] for more details.) 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 validate query response time of our simulator again. We model query response time as a function of query length and term frequency. We model document retrieval time as a fixed time on a first-come-first-serve disk. Our validation demonstrates 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.

3.2 The Parallel IR server

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

3.2.1 Multithreading vs. Multitasking

Since we already had single-threaded servers for uniprocessor machines [5, 6], we implemented a parallel server via multitasking. This version simply uses a light-weight broker and multiple executables of the single-threaded server on the same machine, communicating by message passing [5, 6]. The broker assigns an IR command to one or multiple available processes, depending on the IR command type and the collection partitioning.

Another natural implementation of a 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 libraries [19]. Because threads within a process share the same virtual address space, context switching between threads is less expensive than 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. We compare the performance of the multithreaded version and the multitasking version in Section 4.1.

3.2.2 Simulator and Validation

This section describes the features and their parameters specific to modeling a parallel IR server and its validation. We extend the simulator to model threads, multiple CPUs, and multiple disks. The system parameters also include the original parameters for a single-threaded system [5, 6]: 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 validate our simulator against the mul-

[2] AltaVista claims that its entire Web space is 60 GB and it indexes 30 million Web pages [2].

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

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%
Average		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

tithreaded implementation of a parallel server using InQuery version 3.1.

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 [12], we experiment with a query set that consists of 1000 short queries, with an average of 2 terms per query that mimic those found in the query set down loaded from the Web server for searching the 103rd Congressional Record [12], and use an *observed* query term frequency distribution obtained from their distribution in the Tipster 1 collection and query sets [7].

We vary the arrival rate and the collection size 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 response time, CPU and disk utilization, and determine the largest arrival rate at which the system supports a response time under 10 seconds. We chose 10 seconds arbitrarily as our cutoff point for a reasonable response time. Previous work [10] uses a larger value, up to 40 seconds. Commercial web searchers support response times faster than 10 seconds, but use optimization not implemented in InQuery such as caching query results for frequently executed queries. Unless otherwise stated, we assume the system begins with a cold start where all term and document accesses cause I/O operations. A warm start is when the inverted file is initially in memory.

Validating Query Operation

This section validates query operations for searching a 1 GB database on a multithreaded server with a configuration of a single CPU and disk as the query arrival rate and the number of threads increase on a AlphaServer 2100 5/250. Each thread executes one query.

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. We assume the system begins with a warm start. Positive numbers indicate the simulator overestimates the actual system. The simulator reports response times that are 4.5% slower than the actual system on the average, and range from 3.9% faster to 27.2% 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. For example, for a system with 16 threads, the simulator is from 0.3% faster to 4.1% slower than the actual system. The difference between the simulator and the actual system increases as a function of the number of queries waiting in the queue, because our query model overestimate the query evaluation time for most of queries (see Appendix A). Overall, the simulator matches the actual system closely.

4 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 explore how threading and additional hardware affect system scalability under a variety of workloads and hardware configurations.

4.1 Multithreading vs. Multitasking

This section compares the performance of the multithreaded and multitasking implementations 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 [14]. Each query is simply a sum of the terms, 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 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 no change (0.0%) 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 Tipster query set with an average of 27 terms per query shows the same trend.

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 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. Decreasing 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.

4.2 Hardware/Software Balancing and System Scalability

In this section, we change from using our implementation to using our simulator to explore system scalability with respect to multiple threads, CPUs, and disks as the collection size increases from 1 GB to 16 GB, and thus explore how software and hardware configurations affect system scalability.

We start with a base system that consists of one thread, CPU, and disk. Our base system is disk bound where the disk is a bottleneck. We improve the performance of our IR server through better software: multithreading; and with additional hardware: CPUs and disks. 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. Adding disks improves performance because partitioning the collection across multiple disks introduces a finer-grain execution of IR commands and shifts the balance of the computation from disk to CPU bound. Adding CPUs also improves performance when CPUs are the bottleneck. However if the hardware components are not balanced, additional hardware can degrade performance. In this section, we demonstrate the system scalability using two sets of experiments. In the first set of experiments, we explore the effects of threading on system scalability. In the second set of experiments, we explore the system scalability with increasing the collection size under two disk configurations. When multiple disks exist, we use a round-robin strategy to distribute the collection and its index over disks.

We assume clients arrive as a Poisson process. For each client, the server performs two operations: query evaluation and retrieving the corresponding summaries.

4.2.1 Threading

This section examines how the software structure, i.e., number of threads, affects system scalability.

Figure 2 illustrates how the average response time changes as the number of threads increases. Figure 2(a) illustrates the configuration using 1 CPU for a 1 GB collection on 1 disk, where the disk is a bottleneck. Figure 2(b) illustrates the configuration using 1 CPU for a 1 GB collection distributed on 4 disks, where the CPU is a

Arrival Rate (per min.)	Number of Threads or Processes						
	1	2			3		
		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 Rate (per min.)	Number of Threads or Processes								
	6			9			12		
	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)

Arrival Rate (per min.)	Number of Threads or Processes						
	1	2			3		
		thr	proc	diff	thr	proc	diff
6	1.23	1.15	1.16	-0.8%	1.16	1.16	**0.0%**
30	1.68	1.20	1.30	-8.3%	1.17	1.26	-7.6%
60	2.83	1.46	1.51	-3.4%	1.26	1.40	-11.1%
300	27.74	12.43	13.16	-7.6%	7.45	7.93	-6.4%

Arrival Rate (per min.)	Number of Threads or Processes								
	6			9			12		
	thr	proc	diff	thr	proc	diff	thr	proc	diff
6	1.16	1.15	0.8%	1.15	1.16	-0.8%	1.15	1.16	-0.8%
30	1.15	1.24	-6.1%	1.15	1.20	-4.3%	1.16	1.21	-4.1%
60	1.23	1.30	-5.6%	1.22	1.29	-5.7%	1.23	1.29	-4.8%
300	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)

bottleneck. Figure 2(c) illustrates the configuration using 2 CPU for a 1 GB collection distributed on 4 disks. Figure 2(d) illustrates the configuration using 2 CPU for a 4 GB collection distributed on 4 disks. Figure 2(e) shows CPU and disk utilization at some interested data points in configuration (a) to (d). The box on the top of each figure lists the system parameters for the experiment. Table 1 defines the abbreviations.

In all the configurations, the average response time improves significantly as the number of threads increases until either CPUs or disks are over utilized, as illustrated in Figure 2(a), (b), and (e). Too few threads limits the system's ability to achieve its peak performance. For example in configuration (c) (see Figure 2(c)), using 4 threads only support 120 requests per minutes at which the system supports a response time under 10 seconds, while using 16 threads supports more than 180 requests per minutes under the same hardware configuration. When either the CPU or disk is a bottleneck, the system needs fewer threads to reach its peak performance. When CPUs and disks are well balanced (configuration (c) and (d)), the necessary number of threads is influenized more by the number of disks than the collection size. In both configuration (c) and configuration (d), the system achieves its peak performance using 16 threads. Additional threads do not bring further improvement.

4.2.2 Increasing the collection size

This section examines system scalability and hardware balancing as the collection size increases from 1 GB to 16 GB. In order to examine possible hardware configurations, we consider two disk configurations: fixing the number of disks and adding disks as the collection size increases, and then vary the number of CPUs in each disk configuration.

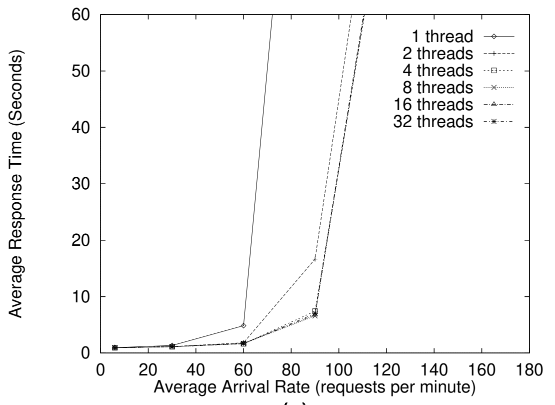
Distributing the collection over a fixed number of disks

In this set of experiments, we fix the number of disks at N and use a round-robin strategy to partition the M GB collection over N disks, where each disk stores all database components for M/N GB of data.

Figure 3 and Table 6 illustrate the average response time and resource utilization when the collection size varies from 1 GB to 16 GB, and are distributed over 16 disks. Each disk thus 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.

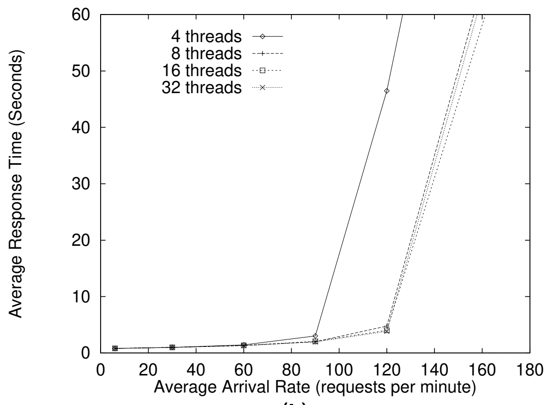
Partitioning the collection over 16 disks illustrates when the system is CPU bound (see Table 6). Although performance degrades as the collection size increases, the degradation is closely related to the CPU utilization. With 1 CPU, the CPU is over utilized for 1 GB and 60 requests per minute. In this configuration, increasing the collec-

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



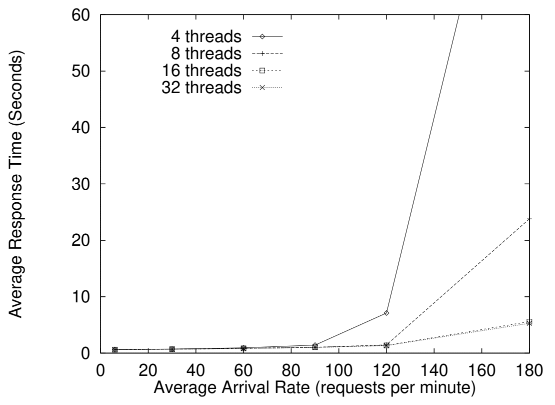
(a)

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



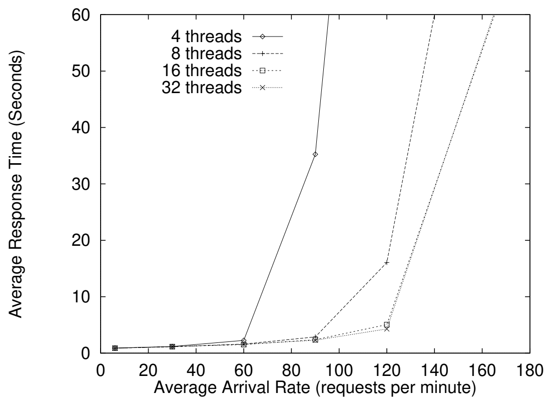
(b)

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



(c)

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



(d)

	Configuration (a)		Configuration (b)		Configuration (c)		Configuration (d)	
	λ = 120		λ = 150		λ = 180		λ = 150	
Resource	TH=2	TH=4	TH=4	TH=8	TH=8	TH=16	TH=8	TH=16
CPU	34.4%	36.2%	72.6%	89.2%	55.1%	56.7%	73.3%	85.2%
DISK	92.7%	97.30%	51.1%	62.8%	77.4%	79.4%	68.0%	79.3%

(e) hardware utilization at some interested data points

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

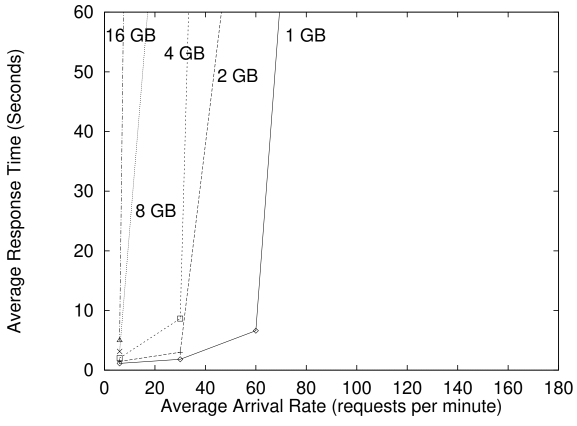
tion size from 1 GB to 16 GB decreases the largest arrival rate at which the system supports a response time under 10 seconds by a factor of 10 (see Figure 3(a)). With 4 CPUs, CPUs are over utilized for 1 GB and 180 requests per minute. In this configuration, the performance degrades much more gracefully (see Figure 3(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 improve 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, or using 4 CPUs for 16 GB; the system supports the arrival rate of 90 requests per minutes, when using 2 CPUs for 2 GB, or using 4 CPUs for 8 GB.

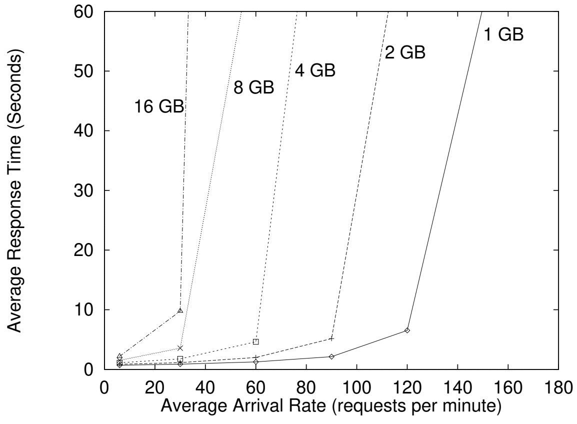
Distributing the collection over a variable number of disks

In this set of experiments, the number of disks increases with the size of the collection. We use a round-robin strat-

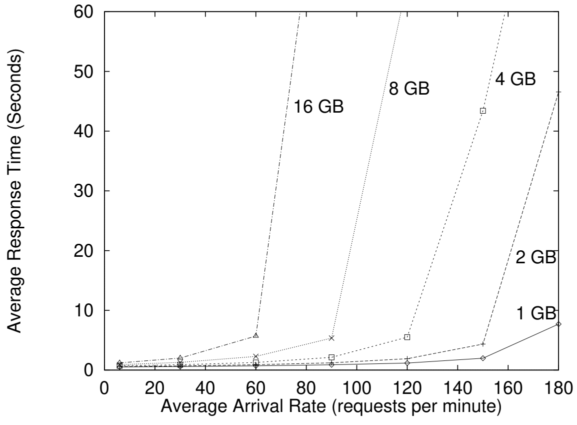
QN	TPQ	QTF	CSIZE	CPU	DISK	TH
1000	2	Obs.	Varied	Varied	16	16



(a) CPU = 1



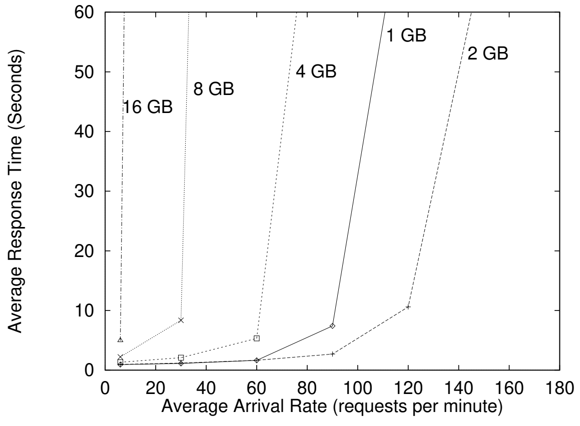
(b) CPU = 2



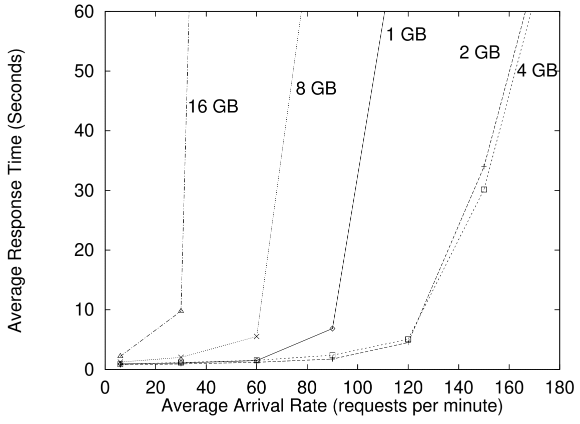
(c) CPU = 4

Figure 3: Average response time for a collection distributed over 16 disks (Simulated)

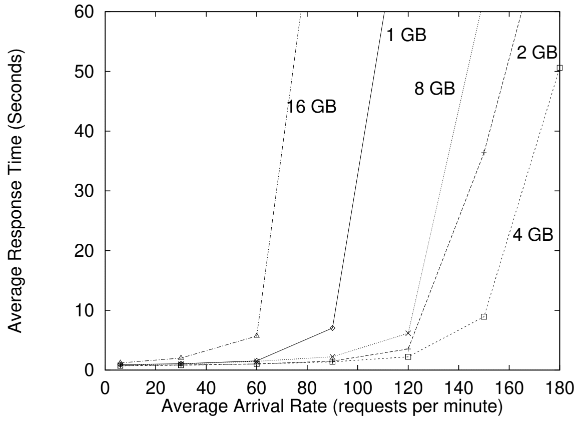
QN	TPQ	QTF	CSIZE	CPU	DISK	TH
1000	2	Obs.	Varied	Varied	Varied	16



(a) CPU = 1



(b) CPU = 2



(c) CPU = 4

Figure 4: Average response time when the number of disks varies with the size of the collection (Simulated)

egy to partition the M GB collection over M disks, where each disk stores all database components for 1 GB of data.

Figure 4 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 the 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 GB collection distributed over 2 disks, the system handles 27.8% more requests than for a 1 GB collection on 1 disk (see Figure 4(a)).

We also observe that adding CPUs does not bring improvement when disks are a bottleneck. For example, a 1 GB collection with 2 and 4 CPUs achieves the same performance as 1 CPU, and a 2 GB collection with 4 CPUs achieves the same performance as 2 CPUs.

Under this disk configuration, we need to double the number of CPUs when we double the amount of data, if we want to support the same level of performance. For example, if we want the system to support an 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.

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 most closely related to the number of disks rather than the collection size. We also find that adding disks degrades system performance, when the CPU is over utilized. For example, for a 2 GB collection, partitioning over 2 disks 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 additional overhead to access each disk. In this configuration, a system with 16 disks performs worse than 2 disks because the CPU is over utilized.

Num.	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

Num.		Size of Collection (size/16 GB per disk)				
CPUs	Resource	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)

Num.	Size of Collection (1 GB per disk)				
CPUs	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

Num.		Size of Collection (1 GB per disk)				
CPUs	Resource	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

However when CPUs are 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 the largest arrival rate supported by a factor of 1.2 compared with partitioning 2 GB over 2 disks (compare Table 5 and Table 7). These results suggest that we need to balance hardware resources carefully in order to achieve scalable performance.

5 Related Work

Although commercial information retrieval systems, such as Web search engines AltaVista and Infoseek, exploit parallelism, parallel computers, and other optimizations to support their services, they have not published their hardware and software configurations that they use to achieve high performance. Although there have been a number of papers regarding to using multiprocessor machines for information retrieval [3, 10, 11, 13, 17, 18, 20, 21, 22], most of them use a distributed memory, massively parallel processing (MPP) architecture [3, 11, 13, 18, 20, 21, 22].

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 [10]. 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. Since we do not have the figures such as clock speed, memory size of their machines, we cannot compare the numbers directly. Our major contribution is not that we can build a faster system, but is that we focus on a single system and analyze how different parameters such as number of threads, disks, and CPUs 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 [17]. 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 use partitioning based on document identifiers in our experiments. We also consider the balance of CPUs and disks, and multiple threads, which are not addressed in this work.

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 [20, 21, 22]. Bailey and Hawking 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 [3]. Cringean *et al.* and Efraimidis *et al.* implement their IR systems on a transputer network, which belongs to the MIMD class of parallel computers [11, 13]. Our work instead uses a SMP and investigates the system performance when processing multiple queries.

6 Conclusion

In this paper, we investigate building a parallel information retrieval server using a symmetric multiprocessor 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 in more detail 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 and utilization, investigate hardware/software balance, and identify system configurations and workloads at which the system supports a response time under 10 seconds. Since our investigation is based on a proven effective, widely used retrieval engine InQuery, our results provide insights for building high performance IR servers for searching on the Web and in other environments using a symmetric multiprocessor.

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 explore information retrieval system scalability with respect to multiple threads, CPUs, and disks as the collection size increases from 1 GB to 16 GB. Our results show that we need more than one thread to fully utilize hardware resources (4 to 16 threads for the configurations we explored). 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. Our results show that we can search more data with no loss in performance in many instances. Although 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.

In summary, because of the mix of CPU and I/O activities, multithreading will significantly improve the throughput and performance of parallel IR servers. Additional CPUs and disks lead to further improvements, but each IR system must explore how to balance them to achieve the best utilization and performance in light of their implementations.

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

References

[1] AltaVista. *http://www.altavista.digital.com.*

[2] AltaVista. About AltaVista search. *http://www.altavista.digital.com/av/content/about.htm.*

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] Douglas Comer. Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] Infoseek. *http://guide.infoseek.com.*

[16] InQuery. An information engine for the U.S. economy. *http://ciir.cs.umass.edu/info/highlights.html.*

[17] 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.

[18] 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.

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

[20] 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.

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

[22] 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.

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

Appendix A: Validation of Basic IR Functionality

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

Query Type	Difference Ave.	Difference Std.	±10%	±20%	±30%	±40%	Ave. Eval. 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
Ave.	2.8%	11.6%	63%	86%	99%	100%	5.1

Table 9: Query Model Validation

Query Type	Total		Distribution		
	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
Ave.	58	39%	43	11	4

Table 10: Distribution of Underestimated Queries

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 [6]. 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, I/O bus, and disk access time. The disk access time accounts for 32% to 90% of the total evaluation time with an average of 73%, and the I/O bus time accounts for 0.4% to 2.5% of the total evaluation time, 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 [6]. 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, I/O bus, and disk access time. The disk access time accounts for 86.7% of the total retrieval time, and the I/O bus time accounts for 0.2%. We represent the summary retrieval time as a sum of the document retrieval times for each document in the summary request [6].

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 of 2 terms per query, 50 medium queries with an average of 12 terms per query, and 50 long queries with an average of 27 terms per query. 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 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 through 7 show the percentage of queries running on the simulator that fall within ±10%, ±20%, ±30% and ±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 through 6 show 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.