Characterization of Real Workloads of Web Search Engines

Huafeng Xi, Jianfeng Zhan*, Zhen Jia, Xuehai Hong, Lei Wang, Lixin Zhang, Ninghui Sun, and Gang Lu
Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China

Abstract

Search is the most heavily used web application in the world and is still growing at an extraordinary rate. Understanding the behaviors of web search engines, therefore, is becoming increasingly important to the design and deployment of data center systems hosting search engines. In this paper, we study three search query traces collected from real world web search engines in three different search service providers. The first part of our study is to uncover the patterns hidden in the query traces by analyzing the variations, frequencies, and locality of query requests. Our analysis reveals that, contradicted to some previous studies, real-world query traces do not *follow well-defined probability models, such as Poisson distribution and log-normal distribution. The second part of our study is to deploy the real query traces and three synthetic traces generated using probability models proposed by other researchers on a Nutch based search engine. The measured performance data from the deployments further confirm that synthetic traces do not accurately reflect the real traces. We develop an evaluation tool that can collect performance metrics on-line with negligible overhead. The performance metrics include average response time, CPU utilization, Disk accesses, and cycles-per-instructions, etc. The third of our study is to compare the search engine with representative benchmarks , namely Gridmix, SPECweb2005, TPC-C, SPECCPU2006, and HPCC, with respect to basic architecture-level characteristics and performance metrics, such as instruction mix, processor pipeline stall breakdown, memory access latency, and disk accesses. The experimental results show that web search engines have a high percentage of load/store instructions, but have good cache/memory performance.*

We hope those results presented in this paper will enable system designers to gain insights on optimizing systems hosting search engines.

*Jianfeng Zhan is the contact person. E-mail: jfzhan@ncic.ac.cn.

1 Introduction

The emergence of popular Internet services, e.g., search, twitter, and social networks, have accelerated a trend toward cloud or datacenter computing [13] [28] [27] Driven by the massive scale of data repositories and the large number of users, these Internet services all require a massive computing infrastructure, which Barroso *et al.* call *Warehouse-scale machines* [13]. A web search engine is a typical example of such services. It is supported not only by search service providers, such as Google, Yahoo, and Baidu, but also by other service providers like social networks and E-Commerce sites. According to the report of comScore, Inc. (http://www.comscore.com/), search has become the most heavily used web mechanism and is an ubiquitous behavior among Internet users. It is thus important for the designers of warehouse-scale machines to understand the characteristics of web search engines.

A web search engine is driven by query requests issued by users over the Internet. In this paper, we call a sequence of time stamped search queries a *workload trace*. In addition, we define a *query* as the whole string of a user request, *a term* as a basic element of a query, a *session* as a group of related queries from the same user.

Due to the difficulty of obtaining real workload traces, many researchers have previously use synthetic traces generated using some probability models [10] [17] [12]. Probability models have been successfully applied to traditional benchmarks, like SPECweb [15] and TPC-C where a workload consists of a series of page requests that can be specified using a *transaction matrix* containing the probabilities of transitions from each given page to other pages. Search engine workload traces largely depend on the (unexpected) behavior of online users. One of the purposes of our investigation is to determine if the commonly used probability models can capture the aspects of a search engine workload. Our analysis of three real search traces reveals that synthetic traces generated according to probability models are susceptible to significant inaccuracy.

Many efforts have been put into analyzing search logs [19] [20] and looking for mechanisms to extract sessions from queries. Those efforts have indeed helped us

better understand the behavior of users of search engines. However, to the best of our knowledge, there has been little work investigating the implications of real user behaviors in the performance of web search engines. In this paper, we use real workload traces from three different search service providers to study these implications.

As part of our effort, we have developed a comprehensive workload characterization tool, named *DCAngel*, which is available from [7]. DCAngel can collect, analyze, and visualize a large number of performance metrics, ranging from performance counters such as cycles-per-instruction and average memory access latency, to quality of services measurements such as the response time of each individual query. It also provides easy-to-use interfaces for users to configure search servers, deploy search engines, and manage search activities online.

Due to the lack of permission to probe real-world web search engines, we set up a search server in our lab using *Nutch* as the search engine, and SoGou web corpus as the indices and snapshot data. However, we have obtained permission to use three real workload traces, one from SoGou [1] and the other two from two of the largest search service providers in China[1]. We have released the search system as a benchmark for datacenter computing, which is named Search and available from [7]. To understand the three traces, we first analyze the variation, frequency, and temporal locality of terms, queries, and sessions, and then attempt to formalize them with two commonly used probability models: the Poisson distribution and the log-normal distribution.

To confirm the analytical results, we replay a real workload trace and three synthetic workload traces to the search engine set up in our lab and measure the resulting performance of each trace. The three synthetic traces are generated from the real traces, but with the query rate variation, request semantics, and temporal locality following probability distributions. The quantitative numbers that we measure include quality of service metrics, architecture-level performance metrics, and OS-level performance metrics.

To help better understand the dynamic behavior of web search engines, we also compare Search with five other types of benchmarks, Gridmix (a MapReduce benchmark), SPECweb2005, TPC-C, SPECcpu2006, and HPCC. We collect the basic architectural metrics, including instruction mix, instruction stall breakdown, and memory access latency for each benchmark. The results show that Search has a number of distinct features, not seen by others benchmarks. In particular, Search has a high percentage of load/store instructions, but has one of the best performances with regarding to the load/store instructions and branch instructions.

[1]Their names cannot be disclosed per the non-disclosure agreements we have signed.

The rest of the paper is organized as follows. In Section. 2, we analyze three real workload traces. in Section. 3, we experiment with both real workload traces and synthetic traces and show system-side differences between performing them. We then present our analysis of architecture-level characterization of Search by comparing it with other benchmarks in Section. 4. Section. 5 describes related work. We conclude in Section. 6 with a summary of our findings and a discussion of future work.

2 Characterization of Real Workload Traces

In this section, we present the results of our study of three real workload traces. As mentioned in Section. 1, for two traces, we cannot disclose their respective source due to legal concerns. For simplicity, we name the three traces Abc, SoGou, and Xyz respectively.

Table 1: Source of Three Workload Traces

	Abc	*SoGou*	*Xyz*
Size	96MB	146MB	194MB
Total Term s	47662	50448	72883
Total Queries	397918	1724264	733444
Duration	72hours	24hours	24hours
Queries/second	1.26	19.9	12.4

Table. 1 shows the duration of the collection period and the number of queries in each trace. Those traces contain only query requests serviced by a single chosen instance of search engines, and do not contain all the query requests received by the web search providers during the duration of collection.

2.1 Methodology

To study the traces, we build a timing model to measure the rate variation of queries and sessions, a semantic model to measure the frequency of queries and terms, a locality model to measure the temporal locality of queries.

Figure.1 shows the control flow diagram of our trace analysis model. The key characteristics of a search workload trace are query sequences and timing sequences. Query sequences depict the contents in each request. Query contents are determined by the semantic model that characterizes the frequencies of terms and the combinations of terms constituting a request. The timing sequences depict the issuing intervals of requests, from which we can compute the fluctuation of query requests. Meanwhile, the order of query requests is determined by the locality model.

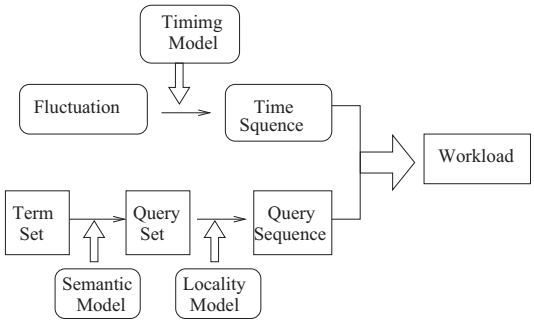


Figure 1: Characterizing a workload trace.

2.2 Timing Model: Rate Variation

The timing model analyzes the time intervals between query requests and attempts to find distribution patterns among intervals. Previous research [17] suggests to use the Poisson distribution to characterize the rate variation of queries. The first attempt of our timing model is to find whether the Poisson distribution can best match a trace. To achieve this goal, we adopt a chi-square goodness of fit test[2] which is used to determine whether sample data are consistent with a hypothesized distribution. The hypothesized distribution here is the Poisson distribution, and hypothesis (H_0) of chi-square goodness of fit is data following the Poisson distribution. The data are divided into k bins and the test static is defined as $\chi^2 = \sum_{i=1}^{k} (O_i - E_i)^2 / E_i$, where O_i is the observed frequency for bin i and E_i is expected frequency for bin i. The hypothesis that the data are from a population with the Poisson distribution is rejected if $\chi^2 > \chi^2_{(\alpha,k-c)}$, where $\chi^2_{(\alpha,k-c)}$ is the chi-square percent point function with $k - c$ degrees of freedom and a significance level of α. So we define a refused number (we call it RN for short) as $RN = \chi^2 - \chi^2_{(\alpha,k-c)}$. We confirm that data follow the Poission distribution if RN is greater than 0, vice versa. To reduce the effect of the uneven distribution of network traffic during a full day (e.g., there are typically a lot more network traffic at early evening than at early morning), we divide each search log into several segments, each of which spans one hour. We analyze the distributions of queries and sessions to check if they fit the Poisson distribution. We adopt a session partitioning algorithm from [20].

Figure.2 shows the refused number's logarithm value of each segment of each trace. For query-level tests, most RN values are greater than zero, which indicates that the rate variation does not follow the Poisson distribution for most of time. The same holds true for session-level tests. One possible explanation of this phenomena is that activities of online users tend to be bursty and have unpredictable randomness or that it is extremely difficult to capture the behavior of online users using one single model.

[2]http://itl.nist.gov/div898/handbook/eda/section3/eda35f.htm

2.3 Semantic Model: Frequency of Terms and Queries

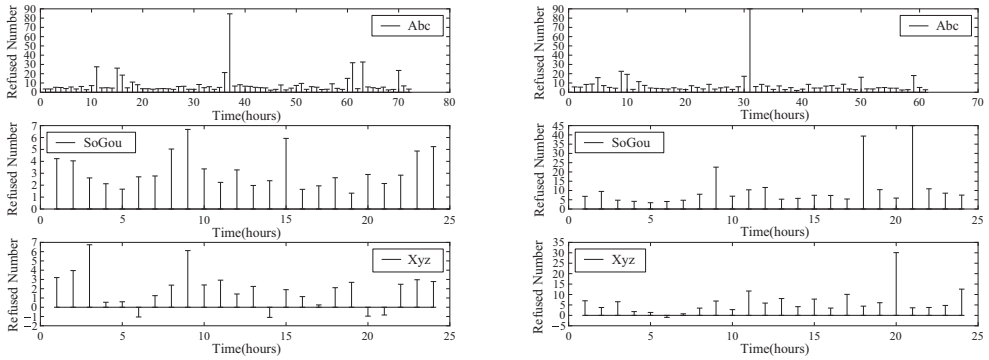
One of the best ways to describe the frequency of terms and queries is Zipf's law. Zipf's law [9] states that given a corpus of natural language utterances, the frequency of any term is inversely proportional to its rank in the frequency table. If the frequency of term A is P and the access count rank of A in a descending order is k, Zipf's law can be described by equation $log(P_a) = C - S \times log(k_a)$, where C and S are constants. Almeida *et al.* [10] demonstrated that the file queries received by a web search server meet Zipf's Law.

We calculate P and k of three workload traces on both the query level and term level. Figure.3 shows the corresponding results. A perfect fit with Zipf's law would show a straight line from the upper-left corner to the lower-right corner. From Figure.3, we can see that the distributions of the frequency of queries follows Zipf's Law almost perfectly for all traces. While there are some small deviations for the distribution of the frequency of terms, we can still consider Zipf's law a reasonable approximation for the frequency of terms.

2.4 Temporal Locality Model of Query

Measuring the temporal locality of a series of queries is similar with measuring the temporal locality of a series of memory accesses. Stack distance [21] is an effective way to analyze the temporal locality of a series of requests. Stack distance, also called reuse distance, depicts the number of different queries between a recurring query. For example, considering a sequence of $\{a_1, b_1, c_1, b_2, a_2\}$, there are three queries a, b, c and the subscript indicates each query's occurrence times. The stack distance of a_2 is two, because there are only two distinct queries between a_1 and a_2: b and c.

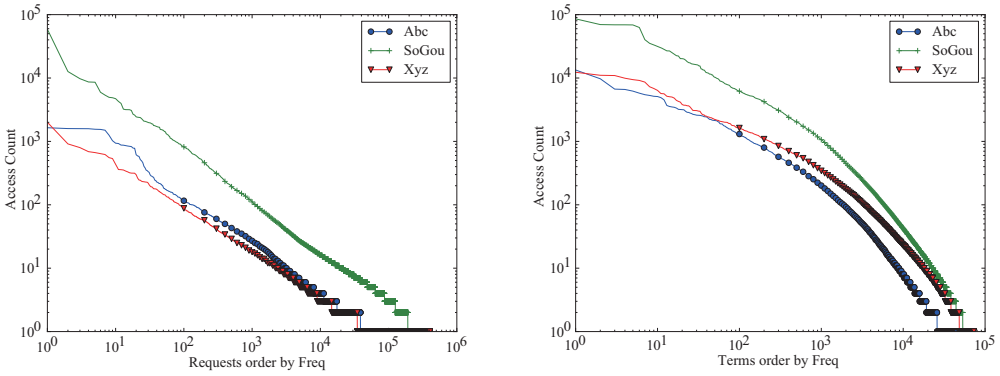
Without losing generality, the temporal locality model assumes the Least Recently Used (LRU) replacement policy and calls the buffer that stores the recently used queries the cache. For a cache capable of saving N recent elements and storing any element in any location (i.e., fully-associative), the LRU algorithm can assure that the elements in the cache are the most recently used N element (Conflict misses in a cache that is not fully associative may affect the effectiveness of LRU. It, however, does not affect the discussion in this paper. A more detailed discussion about conflict cache misses is beyond the scope of this paper.). An element whose stack distance is smaller then N can hit in the



(a) Query-level Poisson Test

(b) Session-level Poisson Test

Figure 2: Poisson Test. Figure.2a and Figure.2b show the refused number's logarithm value, of which the segment span is 1 hour. We confirm that the sample data follow Poisson distribution if the value is greater than 0, vice versa.



(a) Query Frequency

(b) Term Frequency

Figure 3: Zipf's Law on the Frequency of Terms and Queries.

cache.

In [12], Barford *et al.* propose to use the log-normal distribution to approximate the distribution of stack distances among web requests. We investigate if the same log-normal distribution can be used to compute the stack distances for web search queries. Figure.4 shows the miss ratios with varying cache capacities. It clearly shows that none of the real traces performs similarly with a trace with a log-normal distribution.

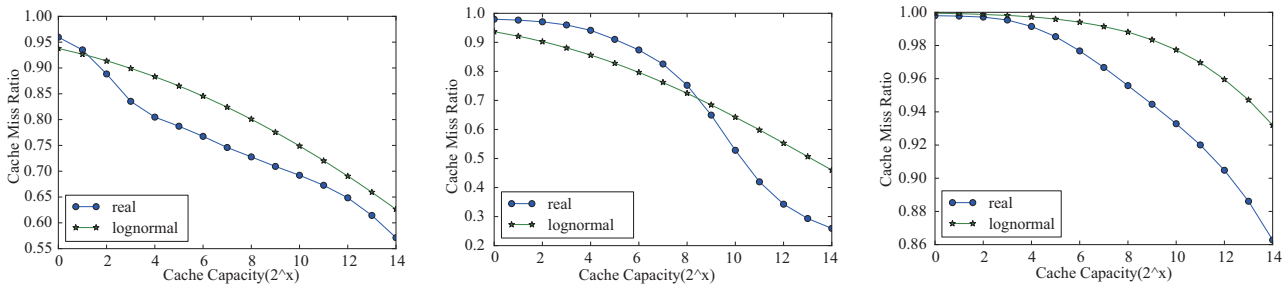
To conclude this section, our analysis shows that while Zipf's law is a good approximation of the frequency of web search terms and queries, the Poisson distribution is an ill fit for the rate variation of web search queries and sessions, and the log-normal distribution is an ill fit for the temporal locality of web search traces.

3 Performance of Real Traces vs. Synthetic Traces

In this section, we attempt to quantify the performance differences between real traces and synthetic traces. We play one real workload trace and three synthetic traces on Searchwith the same snapshot data (i.e., information database). As a point of proof, we use only the trace (from SoGou) here and then generate three synthetic traces from the SoGou trace with a new set of parameters for the query rate variation, frequencies, and temporal locality.

1. The *Poisson trace* keeps the frequencies and orders (temporal locality) of the original real trace, but changes the query rate variation, making it fit with a Poisson distribution.

2. The *hot trace* selects the top 1000 queries according to



(a) Abc (b) SoGou (c) Xyz

Figure 4: Temporal Locality of Queries.

their frequencies from the original trace and then repetitively replay the 1000 queries 100 times. Meanwhile, it changes the query rate variation to fit with a Poisson distribution.

3. The *shuffle trace* keeps the frequencies of the original trace, but changes the distribution of stack distance to fit with a uniform distribution and the query rate to fit with a Poisson distribution.

3.1 Evaluation Methodology

The evaluation environment includes a web (http) server and eight search servers, as shown in Figure.5. Each search server stores a portion of the indices and a portion of the information database in its local file system. The indices and database are from Web Corpus-SoGou [2]. The total size of indices is 16 GB, and the information database contains 32 GB of snapshot data. We split the indices and database equally into eight segments. To achieve improved performance, we distribute two adjacent segments to one search server, e.g., the $(i-1)th$ and $(i)th$ segments are distributed to the $(i)th$ search server (the first server stores the first segment and the last segment). In particular, each search server stores 4 GB of indices and 8 GB of snapshot data.

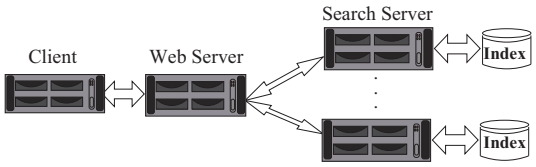


Figure 5: The Architecture of Search

Each web server and search server consists of one Intel Xeon quad-core E5310 processor and 4 GB of memory and runs CentOS 5.5 operating system. Table. 2 lists the important configuration parameters of each server.

Table 2: Details of Configurations

CPU Type		CPU cores	
Intel®Xeon		4cores@1.6G	
TLB	L1 Cache	L2 Cache	Memory
256 entries	32K	4M	4G

When the web server receives a user query, it forwards query to all search servers. Each search server then performs the query and returns a list of documents that match the query. The web server then merges the lists returned from search servers into one list and returns the merged list back to the requesting user.

In order to facilitate our analysis, we have developed a comprehensive workload characterization tool *DCAngel*. DCAngel can collect, analyze, and visualize a large number of performance metrics, ranging from performance counters such as cycles-per-instruction and average memory access latency to quality of services measurements such as the response time of each individual query. It also provides easy-to-use command-line interfaces for users to configure search servers, deploy search engines, and manage search activities online.

DCAngel is implemented in Python and uses SQLite3[3] to manage the collected performance data. SQLite3 allows users to add new functions to extend SQL. In our case, we add a number of statistical functions and visual functions. The statistical functions are used to calculate the standard deviations and correlation coefficients. The visual functions are used to plot different kinds of figures. In addition, we choose matlpotlib as a graphics library for performance data visualization.

Figure.6 shows the high-level diagram of DCAngel. It stores performance data in a relational database managed by SQLite3 that supports the extended SQL statements. Users can access those data through the extended SQL statements.

[3]http://www.sqlite.org/

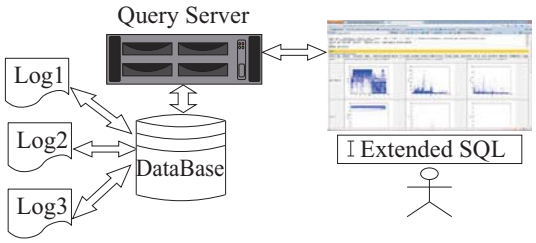


Figure 6: High Level Disgram of DCAngel.

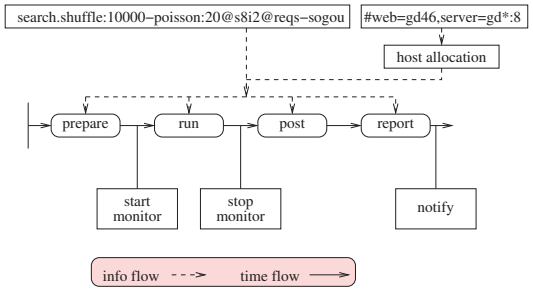


Figure 7: Experiment Procedures.

Figure.7 shows the stages of the evaluation process. It starts with a user input command, which can be divided into two parts. The first part is tagged with the prefix "search..." and contains query information, The second part is tagged with the prefix "#web..." and contains machine configuration. DCAngel is also able to work with the our workload generator tool seamlessly. It interprets input command and calls the workload generator with proper parameters to create the required trace.

After seeing such a command, DCAngel automatically carries out the evaluation stages. As shown in Figure.7, the evaluation process consists of four stages — *prepare*, *run*, *post*, *report*. In the prepare stage, DCAngel distributes data and configuration information to servers. In the run stage, the search engine processes requests coming from the http server and the monitor collects performance data. Once all requests in a trace have been serviced, the monitor stops collecting performance data. In the post stage, performance data collected in the run stage is copied to a designated host for further off-line analysis. In the report stage, the system creates the reports and notifies the users that the experiment has completed.

We use *perf* [6] to collect hardware performance counters, and obtain OS metrics through reading Linux /proc filesystem. The performance metrics we use include three groups: (i) user-observed performance data — average response time and throughput; (ii) OS-level metrics — the average CPU utilization and the average number of DISK I/O operations; and (iii) architectural performance data related to the processor cores and caches.

In addition, we use 10000 queries to warmup the search engine for each run to eliminate the ramp-up effects.

3.2 User-Observed Performance

3.2.1 Response Time

Figure.8 presents the average query response time for the real trace and three synthetic traces. A query response time ($T_{response}$) consists of two parts. The first part is the time interval during which the query stays in the waiting queue (T_{queue}). The other part is the time that the search engine needs to process this query ($T_{service}$).

Relative to three synthetic traces, the real trace has a higher query rate variation. A higher query variation indicates that more query requests are issued in each time interval, resulting in more requests in the waiting queue. Correspondingly, the real trace has a longer response time than others. The rate variation of the synthetic workload traces is similar, so their average response time is mainly decided by the service time. As shown in Section 3.3, for the hot trace that contains only 1000 distinct queries, the web servers are able to keep most of the query results in the caches and the memory. It has the fewest disk accesses and the shortest average response time. The shuffle trace keeps the same set of queries but changes the distribution of stack distance to a uniform distribution, which makes the stack distance of each query larger than the cache capacity. It has the worst locality and longest response time among the synthetic traces.

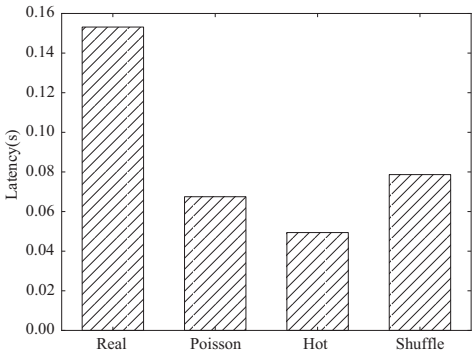


Figure 8: The Average Query Response Time.

3.2.2 Throughput

Figure.9 shows the four different traces' throughput. X-aixs represents average query rate of the whole trace. Y-aixs represents the throughput, which we defined as the number of queries successfully returned to the workload generation tool per second. For each query, we consider it a successful

query only on condition that its response time is less than 1 second. From Figure.9, we can find that the shuffle trace has the worst throughput because it has the worst locality; the Poisson trace has the best throughput because its rate variation is not so severe as that of the real one. The hot trace has a good locality and it consumes the CPU resource more heavily as shown in Figure.10, which has an impact on the throughput.

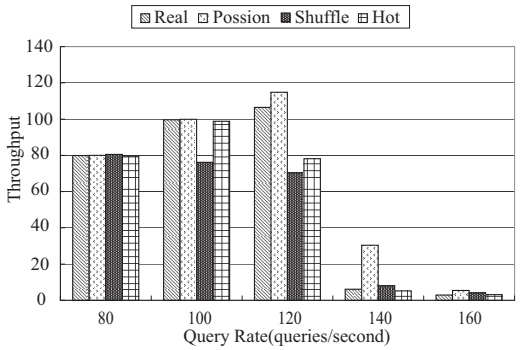


Figure 9: The Throughput of Each Trace.

3.3 OS-level Performance Metrics

Figure.10 presents the cpu utilizations of four workload traces under evaluation. The hot trace has the highest CPU utilization, for two reasons. First, it has good memory performance, as discussed in 3.2, and the CPU rarely has to stop for disk accesses. Second, queries of the hot trace have by average more hits than queries from other traces. Having more hits means more instructions are executed by the search server and the http server, resulting in higher CPU utilization.

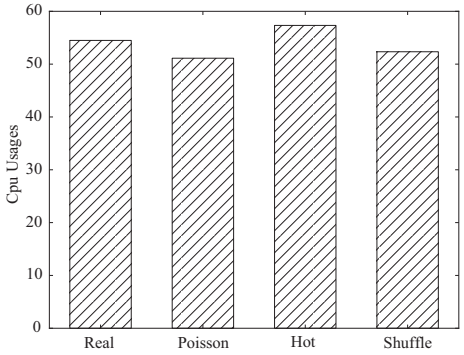


Figure 10: CPU Utilizations.

Figure.11 presents the number of disk sector reads per second. The number of disk accesses is largely dependent on the locality of terms and queries. It is known from the discussion above that the hot trace can effectively use the memory of the search servers and it thus requires very few disk accesses. On the other hand, the shuffle workload has a bad temporal locality because the uniform distribution of the stack distance renders the caching completely ineffective. It has to perform the largest number of disk accesses.

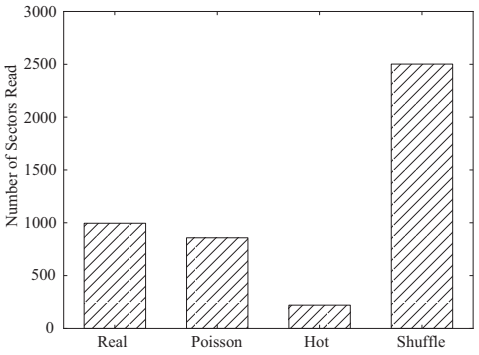


Figure 11: Total Number of Disk Sector Reads.

3.4 Instruction-level Performance Metrics

3.4.1 Instruction Mix

Despite the large differences in performance, all four workload traces have nearly identical instruction mix, as displayed in Figure.12. This phenomena implies that the instruction mix is an inherent feature of the algorithm and implementation of the search engine, is not affected by the (unpredictable) query requests from users.

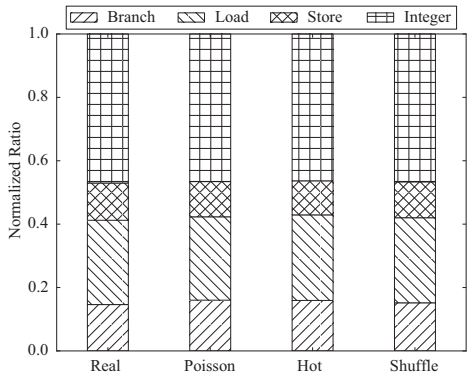


Figure 12: Instruction Mix

3.4.2 Cache Behavior

The miss penalty is the time it takes to replace a block in the upper level of memory hierarchy with the block we need from the lower level[18]. Figure.13 shows the average TLB miss, L1 cache miss, L2 cache miss penalty counted on per-instruction basis. For each level's average miss penalty in Figure.13, we use the following equations to calculate it.

$$MP_L = \frac{N_L \times P_L}{N_{ls}} \quad (1)$$

In Equation (1), The subscript L can be TLB, L1 cache or L2 cache. So MP_L represents the level L's average miss penalty per instruction, and N_L represents the occurrence times of level L's miss. P_L represents the penalty of each L miss happened, and N_{ls} *represents the total number of load and store instructions.* From Figure.13, we can find that all traces have similarly small TLB miss penalty and L1 cache miss penalty. While they all are affected by the L2 cache miss noticeably, their L2 cache miss penalty differs significantly. The shuffle trace has the highest L2 cache miss penalty due to the poor locality caused by uniformly large stack distance. The real trace has high rate variations among its query requests. As different query requests bring different sets of data into the L2 cache, the real trace sees a high L2 cache miss penalty too. The hot trace needs to process more hits per request as a hot query has more matches than the Poisson trace. It consequently has a higher L2 cache penalty than the Poisson trace, which has the lowest L2 cache miss penalty.

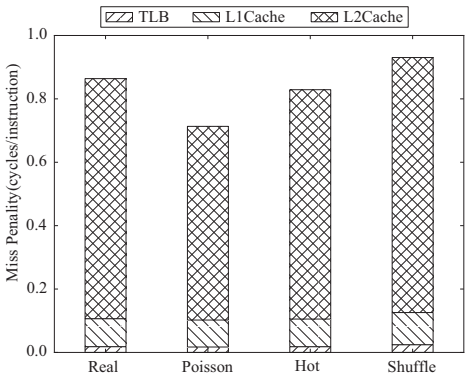


Figure 13: Miss Penalty Per Instructions.

3.4.3 CPI

The CPI of each run is display in Figure.14. Because all traces have near identical instructions mix, they all have near identical percentage of load/store instructions. As a result, the CPI differences of the four traces are mainly decided by the differences in the cache performance.

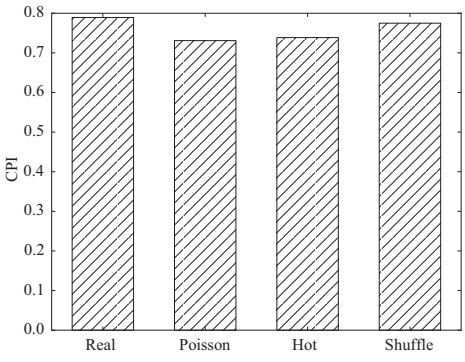


Figure 14: The CPI of each Trace.

To conclude this section, our experimental results show that synthetic traces exhibit very different performance characteristics from real traces with respect to miss penalty per instructions.

4 Architectural characterizations

To put the architectural characteristics of Search into a better perspective, we compare it with five other types of benchmarks. In particular, we consider Gridmix, SPECweb2005, TPC-C, SPECCPU2006, and HPCC in our study. Gridmix [3]is an open source implementation of MapReduce based on Hadoop. SPECweb2005 [15] is a benchmark for evaluating the performance of web servers. TPC-C [4] is an online transaction processing (OLTP) benchmark. SPECCPU2006 [5] is a CPU-intensive benchmark suite, stressing a system's processor, memory subsystem and compiler. HPCC [16] is a set of benchmarks targeting the performance of high performance computing (HPC) systems.

The metrics used to evaluate these benchmarks include instruction mix, processor pipeline stall breakdown, and cache/memory access latency.

4.1 Instruction Mix

We classify all instructions into four categories: load/store, branch, integer operations, and float point/SIMD operations. Please note that some instructions, such as integer or FP operations with indirect memory operands, may be classified into two categories. As a result, the instruction numbers reported here are more close to the number of internal micro operations, not the number of macro architecture instructions. We normalize the total instruction count to 100%. Figure.15 shows the percentage of each category.

We can draw the following observations from Figure.15.

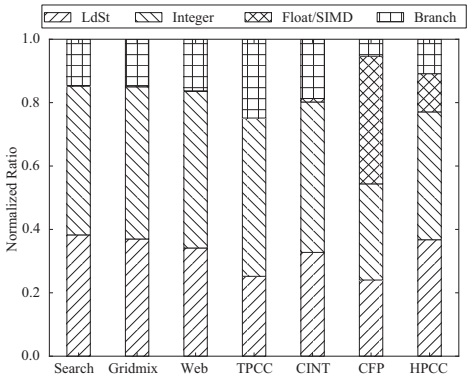


Figure 15: Instruction Mix.

1. The instruction mix of SPEC2006Int/SPEC2006Fp is consistent with what is reported in [24].

2. Only SPEC2006Fp and HPCC benchmarks have Float/SIMD operations.

3. Search has the highest percentage of load/store instructions.

4.2 Cache Performance

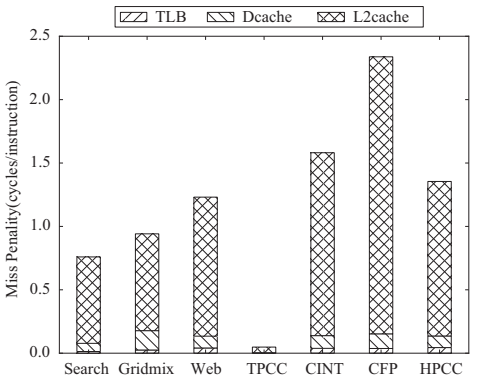


Figure 16: Miss Penalties Per Instruction.

Figure.16 presents the per-instruction latency penalty brought on by TLB misses, L1 cache misses and L2 Cache misses. The method of calculating the miss penalty is the same as mentioned in Section 3.4.2. Without surprises, L2 cache misses have larger impact than L1 cache misses, which have larger impact than TLB misses. In addition, the figure shows that the cache performance of the search engine is second to only TPCC, which indicates that the caches in the processors work pretty well for the web search engine.

4.3 Processor pipeline Stall Breakdown

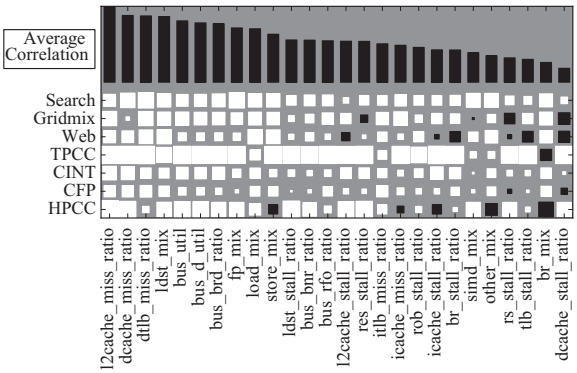


Figure 17: Correlation Coefficients Between CPI and Each of the Architecture-Level Metrics.

In order to decide which factors contribute to processor pipeline stalls, we compute the correlation coefficients between CPI and each of the architecture-level metrics listed in the x-axis of Figure.17. Figure.17 contains the mean coefficients for all benchmarks. Each correlation coefficient in this figure is represented by a small black or white square. A black square means that the coefficient is a negative number. A white square means that the coefficient is a positive number. The area of square is proportional to the absolute value of the corresponding coefficient.

To improve readability of Figure.17, the performance metrics are ranked according to their correlation coefficients in a descending order. The eight highest ranking metrics can be classified into three groups:

1. performance of load/store instructions, including DTLB miss ratio, DCache miss ratio, L2 Cache miss ratio, load/store instructions;

2. performance of the buses, including bus utilization, data bus utilization, and bus burst read ratio;

3. performance of floating point instructions.

The conclusion here is that the performance of these benchmarks is largely decided by the performance of load/store instructions, the buses, and FP instructions.

Figure.18 presents the processor pipeline stalls. From this figure, we can make the following observations.

1. Search has a low percentages of branch stalls, consistent with the fact it has the smallest percentage of branch operations.

2. Search typically has a low percentage of load/store stalls, despite that it has a high percentage of load/store operations. This result is again consistent with the aforementioned good cache performance of Search.

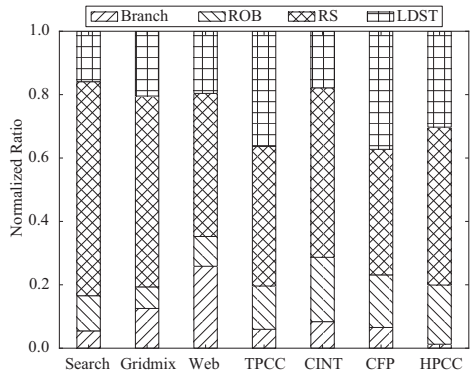


Figure 18: The Breakdown of Processor Pipeline Stalls, where $Branch$ is short for Branch Miss Prediction, ROB is short for Reorder Buffer Full, RS is short for Reservation Station Full, and $LDST$ is short for Load/Store Buffer Full

3. All benchmarks suffer significantly for the shortage of reservation stations, indicating enlarging the reservation stations may be an effective way to gain performance boost for all types of applications.

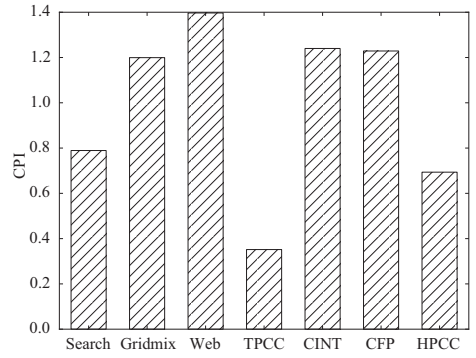


Figure 19: CPI.

Figure.19 presents the CPIs of all benchmarks. It shows that the CPI of Search is smaller than that of other benchmarks but TPCC. This is because Search has almost no FP/SIMD instructions that typically demand longer latency and smaller instruction throughput and has excellent memory access performance as discussed above.

To conclude this section, our experiments demonstrate that Search is rich in load/store instructions but does not contain as many branch instructions as others, and the existing processor caches work well with the web search engine.

5 Related Work

There have been a number of research activities on the web workload generators. Httperf [23] provides a flexible facility for generating various Http workloads and measuring the performance of web servers. It also provides a mechanism to replay the generated traces according to the designated request rate and distribution. Flood [25] is another Http workload generator, used mainly for the saturated performance tests, and it does not provide knobs that allow users to control the time intervals between requests. Surge [12] is a Http workload generator for synthetic traces. It is based on the analytical models of the behavior of web users.

There also have been multiple attempts to generate killer benchmarks for web search engines. SearchGen [11] is a benchmark for the search engine in the context of scientific literature digital library. Its workloads are generated according to an analytical model. Michael *et al.* [22] analyze behaviors of search engine systems with different configuration. They, however, do not compare real workloads with synthetic workloads.

YCSB [14] is Yahoo's Cloud Serving Benchmark framework with the focus on benchmarking the new generation of cloud data serving systems. It also uses synthetic workloads.

Our previous work proposes a precise, scalable and online request tracing for multi-tier services [26], which will further help understand the workload characterization.

6 Conclusion And Future Work

One of the key contributions of our work is that we have used three real query traces collected from different web search service providers. We analyzed the variation, frequency, and temporal locality of terms, queries, and sessions of the real traces. The results demonstrate that neither the Poisson distribution nor the log-normal distribution can accurately capture the key elements of real traces, thus proving that any synthetic traces generated using these distributions are susceptible to significant errors when being used in evaluating the performance of web search engines. However, our study does demonstrate that the frequencies of terms and queries often fit well with Zipf's law.

To confirm the analytical results, we create three synthetic traces from the real traces, using the rate variation, request semantics, and temporal locality enforced by designated probability models. We replay the real trace and synthetic trace onto a search engine in our lab and measured the resulting performance of each trace. The experimental results confirm that the synthetic traces all have significant deviation from real traces with respect to miss penalty per instruction.

We released the web search engine plus the data and the workload trace as a benchmark for datacenter and cloud computing, named Search. To better understand the dynamic behavior of web search engines, we also compare Search with five other types of benchmarks. We collect the basic architectural metrics, including instruction mix, instruction stall breakdown, and TLB/Cache miss penalty for each benchmark. The results show that Search has one of the best performances regarding to the load/store instructions and branch instructions.

We find that real workload, application, and data are all important for characterizing datacenter systems. Internet service companies indeed own big data, and real applications, however they would not like to share data or application with research communities for commercial confidentiality, which is a data lock-in issue. We are being building a testbed for datacenter and cloud computing, which is available from [8]. The testbed will provide real big data, application, and live workloads for both architecture and system research communities.

7 Acknowledgments

We are very grateful to anonymous reviewers. This work is supported by the Chinese 973 project (Grant No.2011CB302500).

References

[1] http://www.sogou.com/.
[2] http://www.sogou/labs/dl/t.html.
[3] http://hadoop.apache.org/mapreduce/docs/current/gridmix.html.
[4] http://www.tpc.org/tpcc.
[5] http://www.spec.org/cpu2006.
[6] Performance counters for linux. https://perf.wiki.kernel.org/index.php/Main_Page.
[7] Search: a benchmark for datacenter and cloud computing. http://prof.ncic.ac.cn/DCBenchmarks.
[8] A testbed for datacenter and cloud computing. http://prof.ncic.ac.cn/htc-testbed.
[9] Zipf's law. http://en.wikipedia.org/wiki/Zipf's_law.
[10] C. Almeida, A. Bestavros, M. Crovella, and A. De Oliveira. Characterizing reference locality in the WWW. In *pdis*, page 0092. Published by the IEEE Computer Society, 1996.
[11] H. Anand and S. Giles. Searchgen: a synthetic workload generator for scientific literature digital libraries and search engines. In *Proceedings of the 7th ACM/IEEE Joint Conference on Digital Libraries: Vancouver, British Columbia, Canada, June 18-23, 2007: building & sustaining the digital environment: JCDL 2007*, page 137. Citeseer, 2007.
[12] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. *ACM SIGMETRICS Performance Evaluation Review*, 26(1):151–160, 1998.
[13] L. Barroso and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis Lectures on Computer Architecture*, 4(1):1–108, 2009.
[14] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.
[15] S. P. E. Corporation. Specweb2005: Spec benchmark for evaluating the performance of world wide web servers. http://www.spec.org/web2005/, 2005.
[16] J. Dongarra, P. Luszczek, and T. U. KNOXVILLE. *Introduction to the HPCChallenge benchmark suite*. Citeseer, 2005.
[17] Ş. Gündüz and M. Özsu. A poisson model for user accesses to web pages. *Computer and Information Sciences-ISCIS 2003*, pages 332–339, 2003.
[18] J. Hennessy, D. Patterson, and D. Goldberg. *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2003.
[19] B. Jansen. Search log analysis: What it is, what's been done, how to do it. *Library & Information Science Research*, 28(3):407–432, 2006.
[20] B. Jansen, A. Spink, C. Blakely, and S. Koshman. Defining a session on Web search engines. *Journal of the American Society for Information Science and Technology*, 58(6):862–871, 2007.
[21] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger. Evaluation techniques in storage hierarchies. *IBM Journal of Research and Development*, 9(2):78–117, 1970.
[22] M. Michael, J. Moreira, D. Shiloach, and R. Wisniewski. Scale-up x scale-out: A case study using nutch/lucene. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 1930.
[23] D. Mosberger and T. Jin. httperf— tool for measuring web server performance. *ACM SIGMETRICS Performance Evaluation Review*, 26(3):31–37, 1998.
[24] A. Phansalkar, A. Joshi, and L. John. Analysis of redundancy and application balance in the spec cpu2006 benchmark suite. *ACM SIGARCH Computer Architecture News*, 35(2):412–423, 2007.
[25] A. H. S. Project. Flood: a profile-driven http load tester. http://httpd.apache.org/test/flood/, Mar. 2011.
[26] B. Sang, J. Zhan, G. Lu, H. Wang, D. Xu, L. Wang, Z. Zhang, and Z. Jia. Precise, scalable, and online request tracing for multi-tier services of black boxes. *IEEE Transactions on Parallel and Distributed Systems*, 2011.
[27] L. Wang, J. Zhan, W. Shi, and Y. Liang. In cloud, can scientific communities benefit from the economies of scale? *IEEE Transactions on Parallel and Distributed Systems*, 2011.
[28] P. Wang, D. Meng, J. Han, J. Zhan, B. Tu, and X. Shi. Transformer: a new paradigm for building data-parallel programming models. *Micro, IEEE*, 30(4):55–64, 2010.