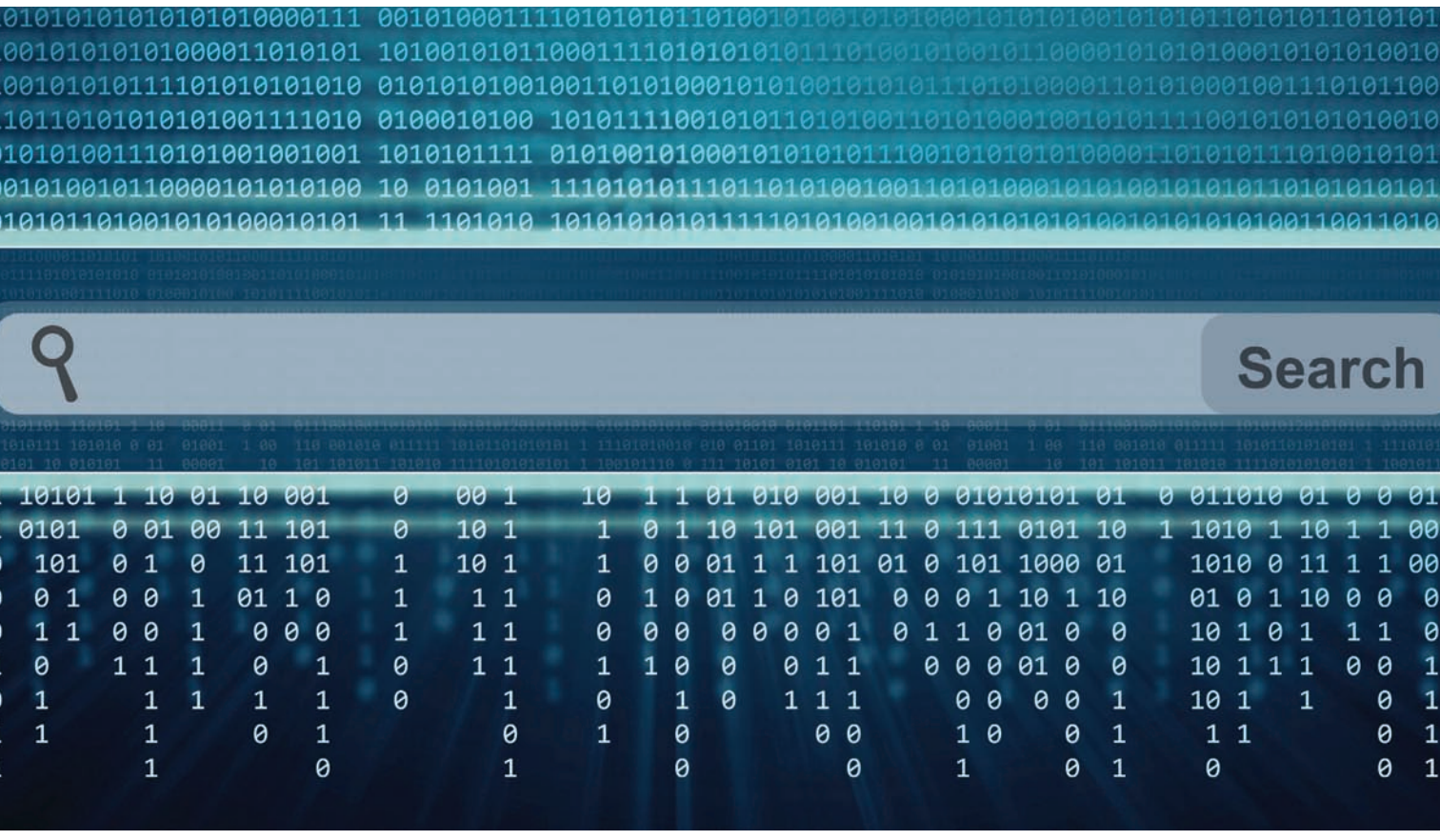
Editors: Barry I. Schneider, bis@nist.gov | Gabriel A. Wainer, gwainer@sce.carleton.ca



Simulating Search Engines

Mauricio Marín | Universidad de Santiago, Chile
Verónica Gil-Costa | Universidad Nacional de San Luis, Argentina
Carolina Bonacic and Alonso Inostrosa | Universidad de Santiago, Chile

Search engines are used by millions of people per second around the world. The term *search engine* refers not only to well-known commercial Web search engines such as Google, Yahoo, and Bing but also to a wide range of search systems that are part of major Web-based applications such as email and social networks. In these applications, they support services such as contextual advertising and special-purpose local search. These systems are made of different algorithms and heuristics whose designs are devoted to optimizing the rate of query requests finished per unit time (query throughput) provided that no single query takes a running time beyond a given upper bound. In addition, the load on computational resources is usually driven by unpredictable user behavior, which takes the form of highly dynamic rates of incoming queries. These constraints must be guaranteed, provided no processor is used beyond a given upper bound so that processors are prepared to cope with sudden peaks in query traffic.

In general terms, large-scale search engines can be seen as multicomponent systems whose individual design, implementation, deployment, and operation are always in constant evolution, and thus it's of paramount importance to be able to predict their performance with precise and practical methods. We've found discrete event simulation to be a useful tool in this context because it enables us to both represent the actual system in a one-to-one correspondence with its main components (including user behavior) and simulate the cost of their relevant operations in a precise and high-level manner. This requires modeling the cost of the different operations involved in processing very large streams of user queries both at macroscopic (cluster of processors) and microscopic (multicore processors) levels.

 1521-9615/17/$33.00 © 2017 IEEE Copublished by the IEEE CS and the AIP

In this article, we describe a methodology to produce such simulations, where, depending on the worldview used to represent and implement the respective simulation model, you can either efficiently obtain the performance metric values for alternative simulated designs or directly assess comparative performance of alternative algorithms by embedding their actual C++ codes into the simulator. To illustrate our proposal, we present microscopic- and macroscopic-level case studies represented with different worldviews: Discrete Event System Specification (DEVS), discrete-event realization of timed colored Petri nets (CPN), and process-oriented simulation (POS).

Search Engines

Web search engines are composed of three main elements: the crawler recovers documents from the Web, the indexer indexes the documents collected by the crawler, and the searcher solves user queries by using the generated index and other components required to achieve efficient performance. Figure 1 shows the relationship among the three elements. In this article, we focus on how to simulate the searcher to evaluate the performance of alternative algorithmic designs for its components. In the searcher, users submit queries composed of keywords, and, in return, they receive a list of pointers to Web documents ordered in accordance with a relevance metric function on the query keywords. We refer to the searcher as the search engine.

A search engine is usually built as a collection of services deployed on a large cluster of processors, wherein each service is distributed onto a set of processors. The processors and the communication network are expected to be constructed from commodity hardware. Each processor is expected to be a multicore processor, enabling efficient multithreading on shared data structures. Message passing is performed among processors to compute on the distributed memory supported by the processors.

The realization of each service can involve the use of different distributed/multithreaded query solution algorithms, distributed query routing algorithms, query results caching policies and heuristics, compressed index data structures, and many other optimizations,[1–6] all devised to enable efficient query processing and whose comparative performance can be evaluated through simulation.[7,8] The key issue is to place the studied strategies in a simulated environment capable of reproducing real operational conditions in a precise manner. This leads to the challenge of properly modeling the relevant running time costs of query-processing strategies and system/hardware software operation.

User queries can travel to several services to be solved. Figure 2 shows an example with three typical services: front-service (FS), caching-service (CS), and index-service (IS). Each query is received by the FS, which redirects it to the CS. The CS then

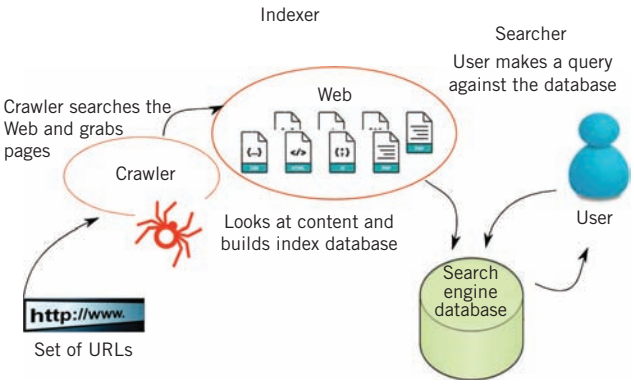


Figure 1. Web search engine overview. Search engines are composed of three main elements: the crawler recovers documents from the Web; the indexer indexes the documents collected by the crawler; and the searcher solves user queries by using the generated index and other components required to achieve efficient performance.

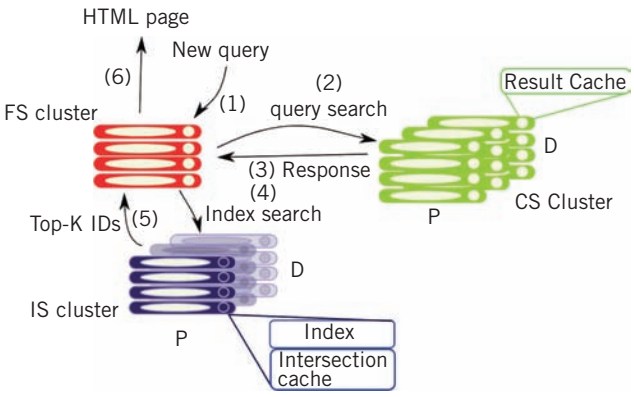


Figure 2. Query processing. Each query is received by the front-service (FS), which redirects it to the caching-service (CS). The CS then checks whether the same query has already been solved and if its results (document IDs) are stored in the service. The CS can answer the FS with either a cache hit or a cache failure. In the latter, the FS sends the query to the index-service (IS), which proceeds to compute the top-K results of the query and sends them back to the FS and CS. The IS uses a document index data structure and additional special-purpose caches to speed up query processing.

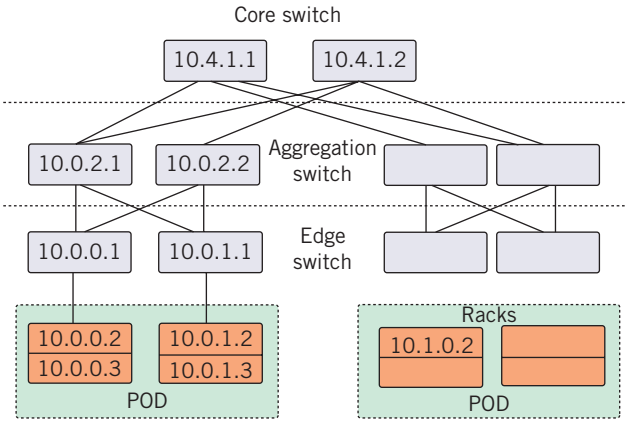


Figure 3. Fat-tree network. At the bottom, processors are connected to edge switches; at the top, we see the so-called core switches; and in the middle, we have the aggregation switches.

checks whether the same query has already been solved and if its results (document IDs) are stored in the service. The CS can answer the FS with either a cache hit or a cache failure. In the latter, the FS sends the query to the IS, which proceeds to compute the top-K results of the query and sends them back to the FS and CS. The IS uses a document index data structure and additional special-purpose caches to speed up query processing.

Search engine services are deployed on arrays of $P \times D$ processors, where P is the level of data partitioning and D is the level of data replication. This architecture uses partitioning to reduce individual query response times and replication to increase the number of queries solved per unit time (query throughput). Replication also provides support for fault tolerance. Notice that, in practice, the upper bounds set for query response times are so small that hitting secondary memory during query processing isn't an option. Thus, all data structures are kept in the processors' main memory in compressed format. At any given time, many different queries can be solved by different replicas of the same partition; in each processor, several queries can be solved in parallel via multithreading.

In the IS, a document index called the inverted index quickly determines the list of documents that contain query terms. After retrieving the list of documents, a ranking algorithm is passed through the document list to determine the top-K documents that are the most pertinent to the query terms. The amount of space occupied by the documents and inverted index is usually huge, so they must be evenly distributed on a large set of distributed memory processors in a sharing nothing fashion. Compression algorithms have been devised for the inverted index and documents.[9] The rationale for the array of processors is as follows: each query is sent to all of the P partitions, and, in parallel, the local top-K document IDs in each partition are determined. These local top-K results are then collected by the requesting FS processor to determine the global top-K document IDs. Alternatively, one of the IS processors (selected in a round-robin manner) can perform the global top-K calculation and send the results to the FS processor.

The inverted index is composed of a vocabulary table (containing the V distinct relevant terms found in the document collection) and a set of posting lists. The posting list for a term stores the identifiers of the documents that contain the term itself along with additional data used for the documents ranking function. To solve a query, a processor must fetch the posting lists for the query terms, compute the intersection among them, and then compute the ranking of the resulting intersection set by using algorithms such as BM25 or WAND.[10] The IS can also keep special-purpose caches such as a posting lists intersection cache devised to reduce the number of re-computation of intersections for popular co-occurring query terms.

Services (processors) communicate through a fast network of switches such as the fat-tree.[11] Figure 3 shows a fat-tree with three levels: at the bottom, processors are connected to edge switches; at the top, we see the so-called core switches; and in the middle, we have the aggregation switches. A property of this network is that it allows achieving a high level of parallelism in message traffic to avoid congestion due to the fact that messages can follow different paths in the fat-tree to reach the same given destination.

Search engines can also be required to support real-time search, a situation in which index content can be updated concurrently with query processing. However, there are potential read/write conflicts in each processor that must be handled with concurrency control algorithms. This problem is exacerbated by the fact that people tend to use a reduced set of very popular terms both in documents and in queries. Figure 4 shows an incoming query Q containing the terms $T2$ and $T3$. The thread in charge of solving the query accesses the inverted index to retrieve the relevant documents for the query and perform the document-ranking operation. At the same time, another thread updates the inverted index with a new document that involves updating the post-

ing lists associated with the terms present in the document. Thus, at a microscopic level, the multithreading performed in each processor of the index service has difficulties on its own merits. Thread programming should optimize the use of the memory hierarchy and prevent read/write conflicts. This is a relevant issue because the performance can be dramatically degraded due to conflicting read and write operations among threads.

In situations where the index isn't updated concurrently, an appropriate number of threads must be scheduled to efficiently solve queries and avoid idle/overloaded threads. A pragmatic approach to multithreaded query processing is to let each incoming query be processed sequentially by a single thread. However, the query running time required to solve some queries containing popular terms can be large because the length of the corresponding posting lists can be quite large as well. One solution is to exploit parallelism by assigning several threads to solve single queries.[12] In this case, to increase the chances of keeping all threads busy doing useful work, query processing can be decomposed into a number of units of work that are placed in an incoming query queue from which threads can take their next job. In the example in Figure 5, threads are dynamically assigned to the processing of individual queries in accordance with their predicted query running times. This prediction is made by using machine learning techniques. Query processing is decomposed into several independent work units that are stored in the shared queue and then processed in parallel by multiple threads. Access to the queue is controlled by a simple locking mechanism. A classification method helps determine the number of work units per query so that the load balance for the parallel query solution isn't compromised. In this particular case, the classification method selects the minimum number of threads so that a target average execution time is satisfied: queries hitting large posting lists are assigned more threads than queries with smaller ones.

Search engines can also rely on a cache hierarchy to enhance performance. Hierarchy goes from the most specific precomputed data for queries (results cache) to the most generic data required to solve queries (posting list cache).[13–15] The result cache (RCache) stores the most frequent queries along with its result webpage; it's deployed on the CS. A location cache (LCache) stores frequent queries not found in the RCache; for each query, it keeps the list of IS partition IDs that provide the query answer. This is a tiny cache as it only stores partition IDs in compressed format. The rationale is that, for frequent queries evicted from the RCache, the LCache can reduce the number of processors involved in the query solution. In addition, under a situation of high query traffic (see Figure 6), the LCache is used as a semantic cache,[16] instructing the FS to send queries to the most promising IS nodes to obtain approximate docu-

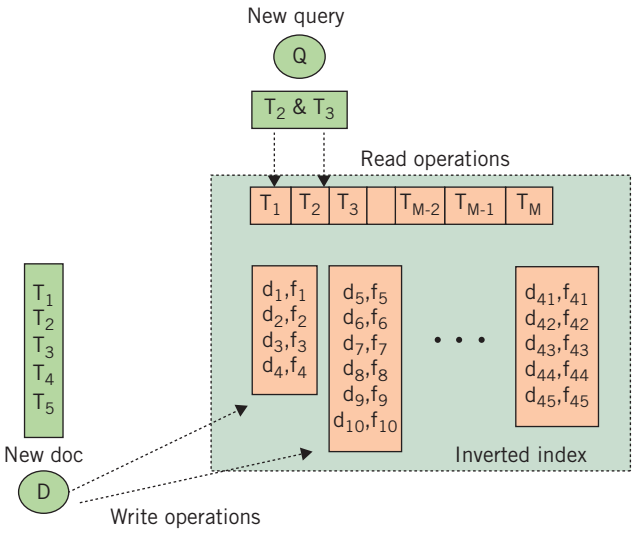


Figure 4. The inverted index. For an incoming query Q containing the terms T2 and T3, the thread in charge of solving the query accesses the inverted index to retrieve the relevant documents for the query and perform the document-ranking operation. At the same time, another thread updates the inverted index with a new document that involves updating the posting lists associated with the terms present in the document.

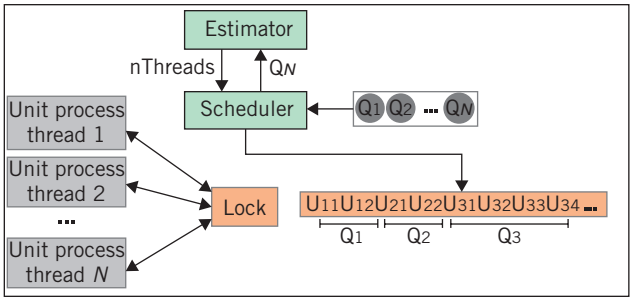


Figure 5. Query work units distribution among threads. Threads are dynamically assigned to the processing of individual queries in accordance with their predicted query running times. Query processing is decomposed into several independent work units that are stored in the shared queue and then processed in parallel by multiple threads.

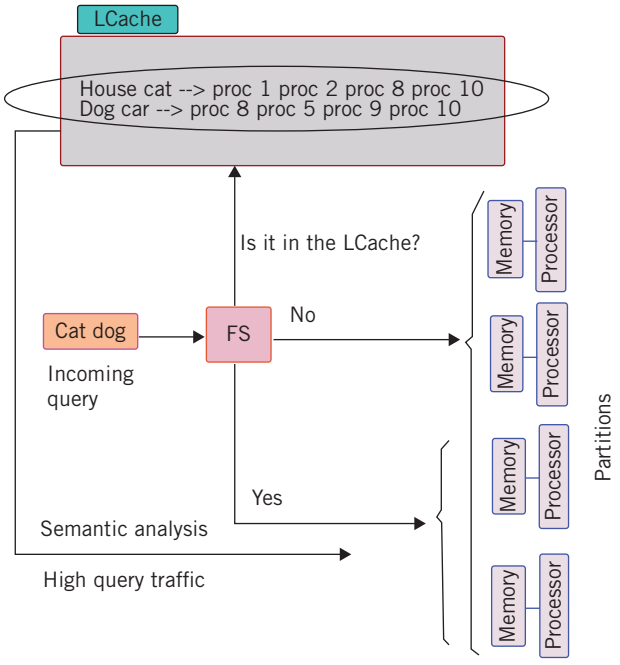


Figure 6. LCache as a semantic cache for high query traffic scenarios. The LCache has the net effect of reducing the average number of processors involved in solving queries, which implies reducing the total amount of hardware required to serve a given workload.

ment results. Overall, the LCache has the net effect of reducing the average number of processors involved in solving queries, which implies reducing the total amount of hardware required to serve a given workload.

Upon a cache hit, the list of documents stored in the cache is used to build the results webpage for the query. In addition, an intersection cache keeps, in each IS node, the intersection of the posting lists associated with pairs of terms frequently occurring in queries. The intersection operation is useful for detecting documents that contain all the query terms, which is a typical requirement for the top-K results in major search engines.

Ultimately, many issues must be taken into consideration to formulate a proper performance prediction model: user behavior represented in streams of queries containing terms of differing popularity; drastic changes in user query traffic when events capture world-wide interest; traffic-dependent latency in query computation and communication; multithreading in cluster-processing nodes and message passing among cluster-processing nodes; and different strategies properly combined to reduce query response time and increase query throughput. In addition to the different caches placed at various points and compressed index data structures, the design can include query-routing strategies, load balance strategies, and multiple variants for document-ranking algorithms. These issues make discrete-event simulation suitable for performance evaluation.

Simulation Modeling

In this article, we present a methodology for modeling and simulating search engines that

- uses models of parallel computation to feature hardware/software system costs together with benchmark programs built on the model rules to measure the respective actual costs;
- uses benchmark programs to measure the cost of relevant operations associated with the different stages of query processing of queries; and
- uses a circulating token approach to simulate the cost of these stages, where each token represents a query that visits a number of concurrent objects devised to cause cost in the simulation time.

We apply the BSP models of parallel computation[17] for a macroscopic representation of the system (cluster of processors level) and Multi-BSP[18] for a microscopic representation of the system (multicore processor level). These models provide a well-defined structure of parallel computation that simplifies the determination of hardware and system software costs, as well as the debugging and verification of simulation programs.

In BSP, the cluster of processors is seen as P processors with local memory, where communication among them is performed via point-to-point messages. Parallel execution is organized as a sequence of supersteps. In each superstep, the processors can only work on local data and send messages to other processors. The end of each superstep is signaled by the barrier synchronization of all processors. At this point, all messages are delivered at their destinations so that processors can only consume them at the start of the next superstep. This organization of parallel computation in supersteps enables integrated consideration of computation, communication, and synchronization from both the algorithmic design and performance evaluation viewpoints.

Benchmark BSP programs are executed to feature the cost of these three components of parallel

computation on the target cluster of processors. In Multi-BSP, the concept of supersteps is extended to the processor cache hierarchy, wherein blocks of data (cache lines) are transferred from main memory to caches close to processor cores, enabling algorithm design to be aware of temporal and spatial locality.

From the simulation modeling viewpoint, the BSP and Multi-BSP models of parallel computation enable a high-level representation of the underlying hardware in which search engine query-processing operations are executed. Once the BSP-based simulation program has been debugged and tested, we remove the barrier synchronization requirement of supersteps, leading to a fully asynchronous message-passing parallel processing system that's close to the actual search engine operation.

In addition, we refine the abstract communication network present in BSP by replacing it with a fat-tree network, in which nodes contain a queuing crossbar model of communication switches. In this case, each switch is modeled by following the same approach as in BSP, which, in general, can be stated as defining a high-level functional view of the component, where we can identify its relevant operations from a collection of specific operations available in the component; defining its individual costs in the simulation time with benchmark programs; and identifying the way they interact with each other during component operation under determined workloads and constraints.

The sequence of operations executed by a search engine such as the one in Figure 2 exemplifies the execution Directed Acyclic Graph (DAG) associated with each query. These DAGs are independent of each other, but they implicitly synchronize when they compete for using cluster resources. We simulate these interactions with resources by modeling them as circulating tokens that form DAGs (fork and join) with nodes visiting queuing networks across the cluster resources. Benchmark programs are built to obtain the costs of relevant operations related to query processing. These costs become the work requests in the DAG nodes associated with each query. In turn, each cost becomes the amount of work requested in the respective resource when access is granted by the queuing strategy.

Using the example in Figure 2, let's further explore this DAG concept. Threads in the simulation model execute primitive operations such as {CPU, COMM, DISK, FORK, JOIN, BEGIN QUERY, END QUERY, HIT RCACHE, UPDATE RCACHE}. Each operation is associated with relevant information such as its running time cost and a message's sender and receiver IDs. Each operation's cost can be given by a constant value such as (CPU 2032), or it can be obtained through a DAG variable as in the case when the cost depends on an intersection cache hit or on a particular query content and ranking algorithm (the latter is annotated as (CPU $WAND) in the respective DAG nodes). Additional DAG commands keep information about the query itself, with the $TERMS command indicating query term identifiers, $SIZE indicating each term's posting list length, and $IDQ the query identifier. Figure 7 shows a typical

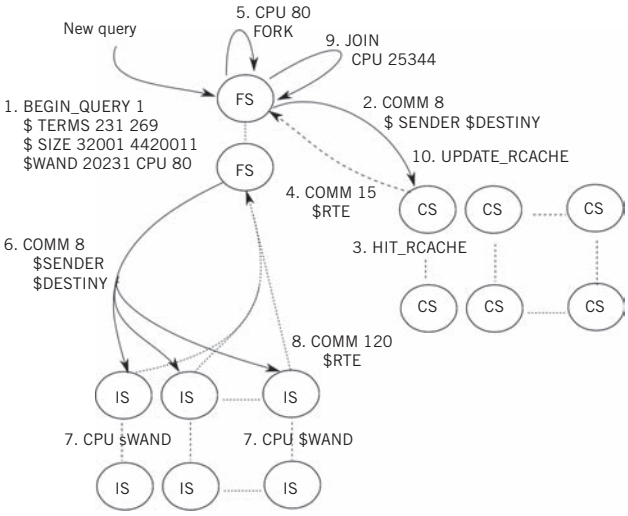


Figure 7. DAG. The advantage of this DAG enhanced with constants, variables, and commands is that it closely resembles the actual steps associated with processing queries in a real search engine.

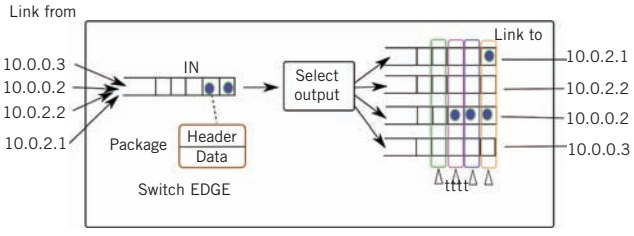


Figure 8. Simulated fat-tree. Messages are divided into packages of fixed size, and each package includes the data, a header with sender and receiver identifiers, and the number of packages forming the message. All input packages go to the same input queue.

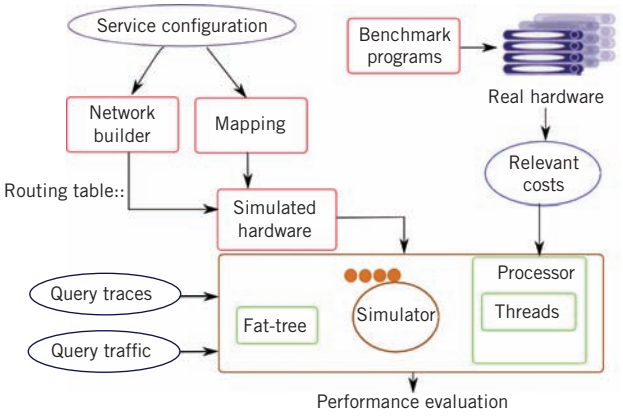


Figure 9. Conceptual model of the proposed methodology. The mapping component assigns services—each service has a unique IP and is linked to the input/output port of the corresponding edge switch to enable point-to-point message passing between processors.

sequence of operations for our example. The advantage of this DAG enhanced with constants, variables, and commands is that it closely resembles the actual steps associated with processing queries in a real search engine.

Figure 8 shows the network simulated as a queuing crossbar network. Messages are divided into packages of fixed size, and each package includes the data, a header with sender and receiver identifiers, and the number of packages forming the message. All input packages go to the same input queue. We keep an output queue (or output port) for each device connected to the switch. Packages forming a message can be sent in parallel through different output queues. If a package is on its way up through the network (from an edge switch to an aggregation switch or from an aggregation switch to a core switch), we evaluate the prefix of the destination IP address to select the output queue.[11] Otherwise, the output queue is selected by evaluating the suffix of the destination IP address. Benchmark programs are devised to evaluate the cost of different communication patterns such as multicast, broadcast, and point-to-point messages.

Methodology and World Views

Figure 9 shows our simulation methodology's main components. The simulation models can be represented with formalisms such as DEVS and CPN. Benchmark programs are based on the BSP and Multi-BSP models and are executed on commodity hardware to determine the costs of relevant operations. For the running example in Figure 2, these benchmarks include processor and network costs, inverted index intersections, and document rankings. The simulation provides three main objects: processors, threads, and a fat-tree network. Each processor is a multicore system enabling efficient multithreading on shared data structures, and message passing is performed through the fat-tree network to enable parallel computation on the distributed memory supported by the processors.

The network builder component creates a fat-tree network with all the links and switches; it also computes the routing table used in the switches. The mapping component assigns services to the processors—that is, each service has a unique IP and is linked to the input/output port of the corresponding edge switch to enable point-to-point message passing between processors.

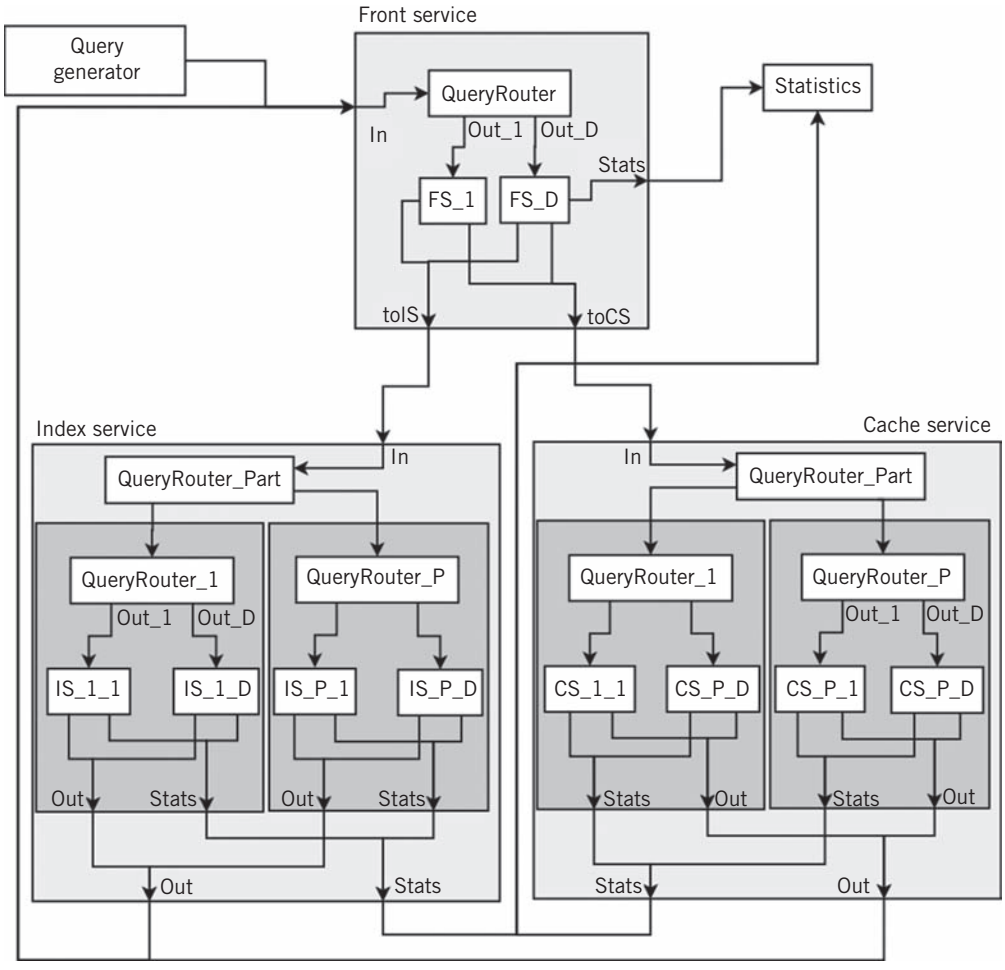
Overall, the proposed simulation methodology enables modeling and simulation at microscopic and macroscopic levels and with different levels of detail. Figure 10 shows an example of a simulation model generated with our methodology represented with the DEVS formalism; Figure 11 shows it with a timed CPN.

Figure 10a shows a macroscopic view of the search engine and its services modeled with DEVS. Each service is a black box with in/out ports connecting with other services and additional elements used to route the queries. Figure 10b shows the formal DEVS definition of the search engine illustrated in Figure 10a.[19] DEVS supports different levels of abstraction through atomic and coupled models. Atomic models are the most elemental and basic entity to represent systems, whereas coupled models represent their structure.

Figure 11a shows a high-level view of the search engine and the FS modeled with CPN. The logic involved at this level is simple and, typically, the models represent the connection between the simulated components and how the query tokens flow. Figure 11b shows a microscopic (low-level) view of threads competing for the resources (places) modeled with CPN. Threads perform read/write operations on the search engine's inverted index (IS processor). In this case, the logic tends to be more complex, and we need to include more detailed information such as the delay and type of data.

Experimental Results

Our experimental results illustrate the effectiveness of our simulation models to predict relevant performance metrics in search engines. As a running



(a)

$WSE = \{X_{wse}, Y_{wse}, D_{wse}, \{ M_{d_{wse}} \; d \in D_{wse}\}, EIC_{wse}, EOC_{wse}, IC_{wse}, select_{wse}\}$
Where:
$X_{wse} = \{\emptyset\}$
$Y_{wse} = \{\emptyset\}$
$D_{wse} = \{\text{QueryGenerator}, FS_i, CS_{jk}\ IS_{lm}\}$
$\forall\ i \in (1, R_{FS}]; \forall\ j \in (1, R_{CS}], \forall\ k \in (1, P_{CS}]; \forall\ i \in [1,R_{IS}], \forall\ m \in [1,\text{-}P_{IS}];$
Where: R_{FS} is the number of nodes in the FS, P_{CS} and R_{CS} are the number of partitions and replicas of the CS, P_{IS} and R_{IS} are the number of partitions and replicas of the IS.
$M_{d_{wse}} = \{ M_{QueryGenerator}, M_{FS_i}, M_{CS_{jk}}, M_{IS_{lm}} \}$
$EIC_{wse} = \{\emptyset\}$
$EOC_{wse} = \{\emptyset\}$
$IC_{wse} \subseteq \{ (\text{(Query Generator, out}_i), (FS_i, \text{in})); ((FS_i, \text{outCS}_{jk}), (\text{CS j k , in})); ((FS_i, \text{outIS}_{lm}), (IS_{lm}, \text{in})); ((CS_{jk}, \text{out}_i), (FS_i, \text{in})); ((IS_{lm}, \text{out}_i), (FS_i, \text{in}));\}$
$select_{wse} = \{\text{QueryGenerator}, FS_i, CS_{jk}, IS_{lm}\}$

(b)

$FS = \{X_{fs}, S_{fs}, Y_{fs}, \delta_{int_{fs}}, \delta_{ext_{fs}}, \lambda, ta\}$
Where:
$X_{fs} = \{(\text{“in”}, q) | q \in Q\}$ is the set of ports and its input values (Query objects).
$S_{fs} = \{\text{“Idle”, “Proc”}\} \times \sigma \times Q \times \text{qSize}$, where “Proc” corresponds to state Processing, $\sigma \in R_0^+$ has the time advance value according to *ta*, and qSize is a variable containing the current amount (size) of queries in the queue with $\text{qSize} \in z_0^+$.
$Y_{fs} = \{(p, q)\ p \in OUT, q \in Q\}$, where q is a Query.

$$\delta_{int_{fs}}(\text{“Proc”}, \sigma, \text{qSize}) = \begin{cases} (\text{“Proc”}, 0, \text{qSize} - 1) & \text{If } 0 < \text{qSize} \\ (\text{“Idle”}, \infty) & \text{o. w.} \end{cases}$$

$\delta_{ext_{fs}}(\text{“Idle”}, a, \text{qSize}) = (\text{“Proc”}, 0, 1)$
$\delta_{ext_{fs}}(\text{“Proc”}, a, \text{qSize}) = (\text{“Proc”}, \sigma - e, \text{qSize} + 1)$
$\lambda(\text{“Proc”}, \sigma, \text{Query}) = (out^*, \text{Query})$
$ta(\text{“Proc”}, \sigma, \text{Query}) = \sigma$

Figure 10. The proposed simulation methodology: DEVS (a) model for a search engine and (b) formalism. On the left, we see a macroscopic-level view of the search engine, and on the right, a microscopic-level view of an FS.

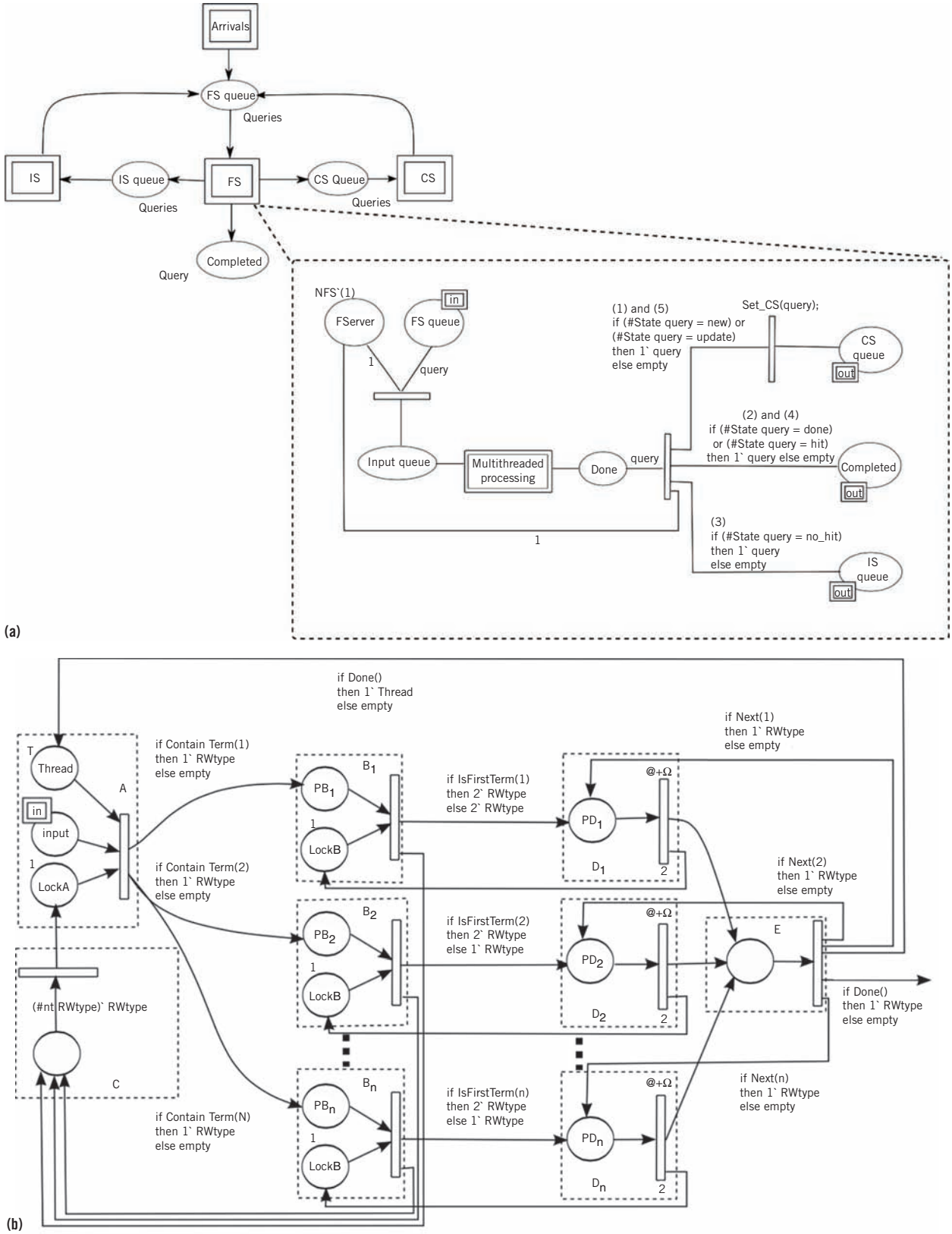


Figure 11. Colored Petri net (CPN) model for a search engine: (a) macroscopic and (b) microscopic modeling levels.

example, we use the search engine described in Figure 2. For this system, we developed both simulators and an actual implementation based on C++ and the MPI message-passing library. The simulators were built using process-oriented simulation (POS) implemented using C++ libCppSim; DEVS, which is run on its own simulation kernel called PCD++[20]; and timed CPN with simulation kernel implemented in C++.

We performed experiments with a log of actual user queries submitted to the AOL search service between 1 March and 31 May 2006. This log, exposing typical patterns present in user queries, has 16,900,873 queries, with 6,614,176 unique queries and a vocabulary consisting of 1,069,700 distinct query terms. We also used a sample (1.5 Tbytes) of the UK Web obtained in 2005 via the Yahoo search engine, over which we constructed a 26-million-term and 56,153,990-document inverted index. We executed the queries against this index to get the cost of query-processing operations. The document ranking method we used was WAND (index service) and the cache policy was LRU (caching service). We ran all benchmarks with the data stored in main memory to avoid access to the disk. The results with the real implementation of the search engine as described in Figure 2 were obtained on a cluster with 256 nodes composed of a 32-core AMD Opteron 2.1-GHz Magny Cours processor with 16 Gbytes of main memory per node. We executed simulations with the same stream of query terms but with additional information such as ranking cost and posting list sizes.

Figure 12 presents results predicting the total number of queries solved per unit time in Figure 2's search engine for different incoming query rates so that the total number of processors holding the FS, CS, and IS nodes are adjusted to the minimum value required to serve the incoming query rate without falling into processor saturation. In all cases, the simulation results achieve good agreement with the results from the actual implementation. The differences are mainly due to the model used for the communication network supporting message passing among processors. In the case of CPN, the network is an oversimplification of the fat-tree network, whereas POS and DEVS implement a more detailed model of the fat-tree network.

Figure 13 shows results for the average running time required to solve individual queries for a different number of partitions of the IS. In this case, simulation results are even closer to those from the actual implementation than those shown in Figure 12. In Figure 13, the incoming queries rate is kept at a value that ensures no processor gets a utilization value beyond 50 percent. This indicates a steady-state operation situation in which the processors have the capacity to respond to a sudden increase in the incoming queries rate.

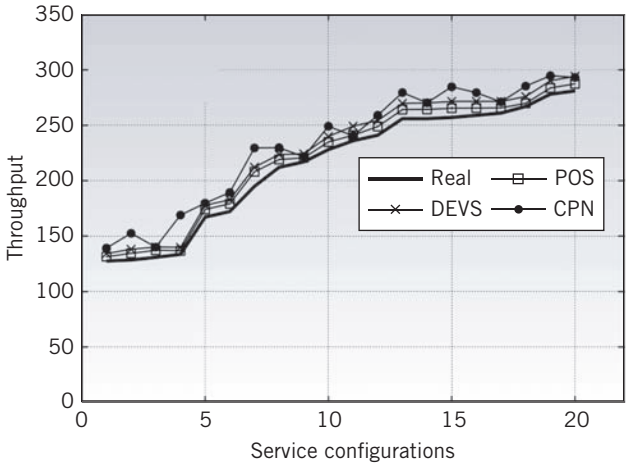


Figure 12. Query throughput results (*y*-axis) achieved with a real implementation and respective simulators of the system presented in Figure 2. The service configurations (*x*-axis) shows the number of processors supporting the FS, CS, and IS nodes ranging from 115 to 240.

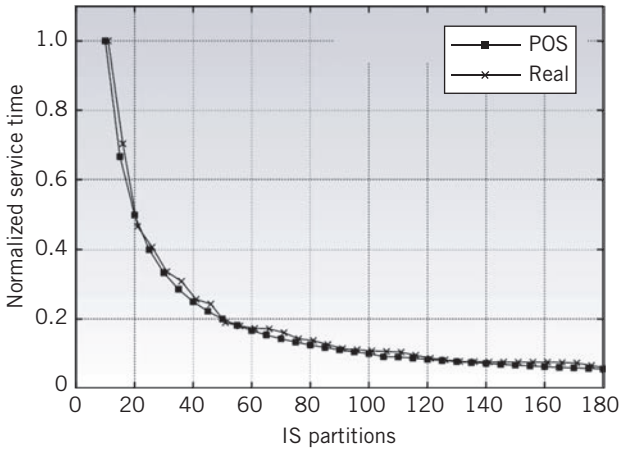


Figure 13. Average response time for individual queries (y-axis) achieved with a real implementation and process-oriented simulator (POS) for the IS in the system in Figure 2. The results are obtained for different numbers of partitions of the IS (x-axis), with each partition held by a single multithreaded processor.

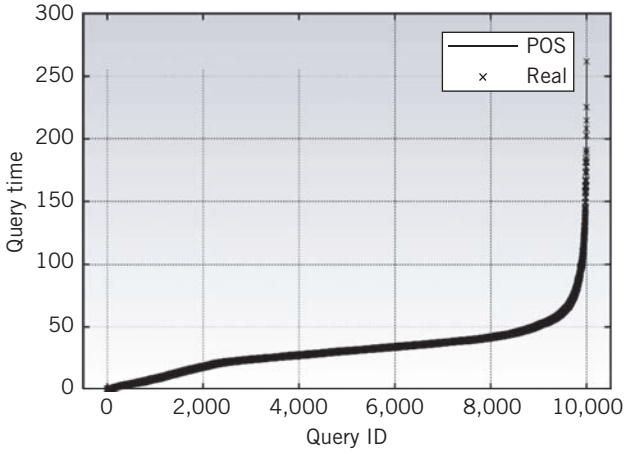


Figure 14. Results for multithreaded query processing with the processor operating at full capacity and using the thread-scheduling strategy described in Figure 5. The *y*-axis presents the response time of individual queries, and the *x*-axis shows the queries ordered by increasing time values.

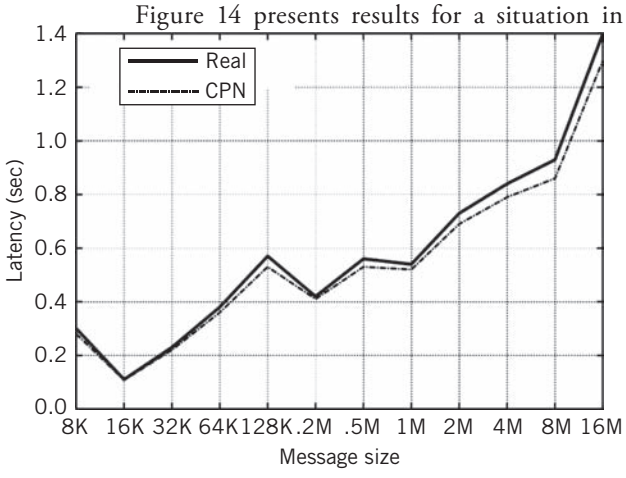


Figure 15. Latency in communication in the fat-tree network connecting the cluster processors for all-to-all message patterns. Specifically, we see results obtained with a benchmark program (curve Real) and a simulator of the fat-tree constructed with timed CPN (curve CPN). The simulation model is described in Figure 8.

Figure 14 presents results for a situation in which a single IS processor is almost 100 percent utilized, meaning working at full capacity. All threads are busy processing queries at all times, with some queries processed in parallel by using several threads and others processed by single threads. This is in accordance with a thread assignment policy devised to achieve a target average value for query response time. Figure 14 shows the response time results for individual queries. These results show that the POS can predict results from the actual implementation in a very precise manner (the observed error is below 1 percent in all cases).

Finally, Figure 15 compares real and simulation results for the communication cost in the fat-tree network. In this case, the model is subjected to increasingly demanding message traffic represented by message sizes. The communication pattern is all-to-all, that is, all processors repeatedly send a copy of a different local message to all other processors. The results show that the CPN simulator can predict the general trend in communication cost at a low error rate. In particular, the root-mean-square error of the deviation, which is a measure of the differences between values obtained by the real benchmark program and the values reported by the simulator, is below 0.35 percent, whereas the relative error is below 1.03 percent. Similar error and trend prediction is observed for the broadcast and unicast communication patterns.

Practice and experience on the design of efficient query-processing strategies for search engines have shown that discrete-event simulation can be a powerful tool for comparing alternative approaches under complex performance metrics. In this application domain, the entire simulation is reduced to emulating competition for using hardware resources, which simplifies performance evaluation under a wide range of possible user query dynamics. A challenging task is to let these simulations execute event processing in parallel to reduce overall running time. The causal relationships among events associated with DAGs poses difficulties to well-known synchronization protocols for parallel discrete-event simulation. Nevertheless, we anticipate that efficient performance is feasible from the fact that competing DAGs usually aren't expected to access the resources in any particular order. This enables relaxation of strict event causality across processors that can be exploited to design efficient optimistic simulations capable of producing approximate but precise enough performance metric values. We plan to further develop this idea in the near future. ■

Acknowledgments

This work has been partially funded by CONICYT Basal funds FB0001 and FONDEF IDeA ID15I10560.

References

1. S. Alici et al., "Adaptive Time-to-Live Strategies for Query Result Caching in Web Search Engines," *Advances in Information Retrieval*, 2012, pp. 401–412.
2. D. Arroyuelo et al., "Document Identifier Reassignment and Run-Length-Compressed Inverted Indexes for Improved Search Performance," *Proc. Special Interest Group Information Retrieval*, 2013, pp. 173–182.
3. R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval: The Concepts and Technology behind Search*, Addison-Wesley Professional, 2011.
4. K. Chakrabarti, S. Chaudhuri, and V. Ganti, "Interval-Based Pruning for Top-K Processing over Compressed Lists," *Proc. Int'l Conf. Data Eng.*, 2011, pp. 709–720.
5. S. Ding and T. Suel, "Faster Top-K Document Retrieval Using Block-Max Indexes," *Proc. Special Interest Group Information Retrieval*, 2011, pp. 993–1002.
6. C. Macdonald, N. Tonellotto, and I. Ounis, "Learning to Predict Response Times for Online Query Scheduling," *Proc. Special Interest Group Information Retrieval*, 2012, pp. 621–630.
7. V. Formoso et al., "Analysis of Performance Evaluation Techniques for Large Scale Information Retrieval," *Proc. Large-Scale Distributed Systems for Information Retrieval*, 2013, pp. 215–226.
8. V. Gil-Costa et al., "Modelling Search Engines Performance Using Coloured Petri Nets," *Fundamenta Informaticae*, 2014, pp. 1–28.
9. H. Yan, S. Ding, and T. Suel, "Inverted Index Compression and Query Processing with Optimized Document Ordering," *Proc. World Wide Web Conf.*, 2009, pp. 401–410.
10. A.Z. Broder et al., "Efficient Query Evaluation Using a Two-Level Retrieval Process," *Proc. Conf. Information and Knowledge Management*, 2003, pp. 426–434.
11. M. Al-Fares, A. Loukissas, and A. Vahdat, "A Scalable, Commodity Data Center Network Architecture," *Proc. Special Interest Group on Data Communication*, 2008, pp. 38:63–38:74.
12. C. Bonacic et al., "Multithreaded Processing in Dynamic Inverted Indexes for Web Search Engines," *Proc. Large-Scale Distributed Systems for Information Retrieval*, 2015, pp. 15–20.
13. T. Fagni et al., "Boosting the Performance of Web Search Engines: Caching and Prefetching Query Results by Exploiting Historical Usage Data," *ACM Trans. Information Systems*, vol. 24, no. 1, 2006, pp. 51–78.
14. Q. Gan and T. Suel, "Improved Techniques for Result Caching in Web Search Engines," *Proc. World Wide Web Conf.*, 2009, pp. 431–440.
15. M. Marin, V. Gil-Costa, and C. Gomez-Pantoja, "New Caching Techniques for Web Search Engines," *Proc. Int'l Symp. High-Performance Parallel and Distributed Computing*, 2010, pp. 215–226.
16. F. Ferrarotti, M. Marin, and M. Mendoza, "A Last-Resort Semantic Cache for Web Queries," *Proc. String Processing and Information Retrieval*, 2009, pp. 310–321.
17. L. Valiant, "A Bridging Model for Parallel Computation," *Comm. ACM*, vol. 33, no. 8, 1990, pp. 103–111.
18. L. Valiant, "A Bridging Model for Multi-Core Computing," *J. Computer and System Science*, vol. 77, no. 1, 2011, pp. 154–166.
19. A. Inostrosa-Psijas et al., "DEVS Modeling of Large Scale Web Search Engines," *Proc. Winter Simulation Conf.*, 2014, pp. 3060–3071.
20. Q. Liu and G. Wainer, "Parallel Environment for Devs and Cell-Devs Models," *Simulation*, vol. 6, no. 83, 2007, pp. 449–471.

Mauricio Marín is a professor at Universidad de Santiago, Chile. His research interests include parallel computing, information retrieval, and discrete simulation. Contact him at mauricio.marin@usach.cl.

Verónica Gil-Costa is an associate professor at Universidad Nacional de San Luis, Argentina. Her research interests include simulation, similarity search and parallel computing, and distributed systems with applications in Web search engines. Contact her at gvcosta@unsl.edu.ar.

Carolina Bonacic is an assistant professor at Universidad de Santiago, Chile. Her research interests include parallel processing on multithreaded processors, information retrieval, and simulation. Contact her at carolina.bonacic@usach.cl.

Alonso Inostrosa is a postdoctoral researcher at Universidad de Santiago, Chile. His research interests include simulation, parallel, and distributed computing. Contact him at alonso.inostrosa@usach.cl.