A Massively Parallel Knowledge-Base Server Using a Hypercube Multiprocessor *

Frank Dehne
Center for Parallel and Distributed Computing, School of Computer Science, Carleton University, Ottawa, Canada K1S 5B6

Afonso G. Ferreira
Laboratoire de l'Informatique du Parallélisme - IMAG, Ecole Normale Supérieure de Lyon, 69364 Lyon cedex 07, France

Andrew Rau-Chaplin
Center for Parallel and Distributed Computing, School of Computer Science, Carleton University, Ottawa, Canada K1S 5B6

Abstract

In this paper we study the parallel implementation of a traditional frame-based knowledge representation system for a general purpose massively parallel hypercube architecture (such as the Connection Machine). We show that, using a widely available parallel system (instead of a special purpose architecture), it is possible to provide multiple users with efficient shared access to a large scale knowledge base. Parallel algorithms are presented for answering multiple inference, assert and retract queries on both, single and multiple inheritance hierarchies. In addition to theoretical time complexity analysis, empirical results obtained from extensive testing of a prototype implementation are presented.

1 Introduction

As outlined in [9, 27, 28], massively parallel architectures are essential for computationally intensive AI applications. Since knowledge representation is an essential part of AI [22, 29], several researchers have studied parallel architectures for implementing knowledge bases [1, 9, 11, 12, 16, 18, 20, 23, 24, 27, 28]. The parallel knowledge representation systems presented in the literature have, however, either been based on special purpose parallel architectures or support only the parallelization of one query at a time. The latter implies the (economically infeasible) dedication of a massively parallel computer to one single user (e.g. [11, 18]).

This paper is concerned with the design and implementation of a parallel frame-based knowledge representation system which allows parallel access to a large scale knowledge base server; see Figure 1.

We consider a parallel implementation of a standard frame based knowledge representation system which allows elementary queries such as top-down and bottom-up inference queries and assert/retract queries [11]. Such a system could be utilized as the foundational layers of a truly parallel reasoning system, see Figure 2. That is, it could be used as a parallel implementation of Layers 1 and 2 in Figure 2. Parallelization of the higher level layers has already been extensively studied [10, 14, 23]. This could lead to an architecture where Layers 3 and 4 reside on multiple workstations being connected to a central SIMD hypercube multiprocessor which supports layers 1 and 2.

More specifically, we show in this paper how to execute in parallel a set of inference and assert/retract queries on a knowledge base (with a frames) stored on a fine grained SIMD hypercube multiprocessor (with N= processors).

In Section 3 we study single inheritance hierarchies with implicit storage [26]. We consider, in Section 3.2, multiple bottom-up inference queries on single inheritance hierarchies. We show that wcN bottom-up inference queries [11] can be answered, in parallel, in time $O(k log_2 n + n + n^2)$ for $O(k log_2 n log_2 n)$ if frames can have an unbounded number of children, where $k$ is the height of the inheritance hierarchy. In Section 3.3, we present a heuristic algorithm for answering multiple top-down inference queries [11]. Our experimental results, obtained from extensive testing of a prototype implementation, show that a nearly optimal (100%) processor utilization is obtained for a 70% load factor (number of processors divided by number of queries). In our experiments, the utilization never dropped below 75%, regardless of the load factor and other parameters. Our system adapts flexibly and automatically to varying work loads in a close to optimal way (providing a nearly constant product of response time and number of queries). In Section 3.5 we study assert and retract queries, and show that they can be executed in essentially the same time complexity as top-down inference queries. Note that, our system can process all four kinds of queries simultaneously, and that only direct communication between adjacent processors is used in our algorithms.

In Section 4 we generalize our results to multiple inheritance hierarchies with explicit storage [26]. We outline how multiple top-down inference queries, bottom-up inference queries, as well as assert and retract queries can be answered in parallel for a multiple inheritance hierarchy stored on a hypercube multiprocessor. The time complexities for these operations are at most a $O(k log_2 n)$ factor larger than the complexities of the respective operations on our parallel single inheritance system.

2 Preliminaries

2.1 Frame-Based Knowledge Representation

Semantic nets alternatively known as frame-based systems have been widely studied [1, 2, 3, 9, 11, 13, 17, 19, 24, 25].

---

* The first author's research is partially supported by the Natural Sciences and Engineering Research Council of Canada (Grant A9173). The second author is currently on leave from the University of Sao Paulo (Brazil), project BID/USP, and partially supported by CAPES/COPPE/CUB. The third author's research is partially supported by the Bolt-Baynor Research Graduate Award Program. Hardware support from the Northeast Parallel Architecture Center (Syracuse, NY) is gratefully acknowledged.

---

Note: In contrast to [11], our time complexity results also account for the inter processor communication time.

---

26), and several general purpose knowledge representation tools have been designed based on them [2, 3, 11, 13, 17, 25]. There are many advantages to a frame based approach for knowledge representation as detailed in [13, 19].

A frame language provides the designer of knowledge based systems with an easy way to describe the domain objects to be modeled and their relationships. In a frame-based system each frame is used to describe an individual object or a class of objects. For example in Figure 3, the class Ocean-Liner is used as a prototype to describe all of the properties that are common to all Ocean liners, such as the fact that they carry paying passengers. The instance QE2, on the other hand, is a frame that represents an individual instance of the class ocean-liner. It specifies knowledge about a particular ocean liner, the QE2, such knowledge might include the number of passengers that the liner carries, or the QE2’s transatlantic crossing time.

Both classes and instances are represented by frames. Each frame consists of a series of slots, where each slot is used to represent a single fact about a particular class or instance. Some slots may be explicit while others may inherit their values from their predecessors in the hierarchy (implicit).

In this paper we will focus first on parallel single inheritance frame based systems. In such systems the inheritance hierarchy can be represented by a k-nary tree. Later, in Section 4, we will sketch how our approach can be extended to handle multiple inheritance; i.e., the inheritance hierarchy has a more general lattice structure.

2.1 Hypercube Multiprocessor

A hypercube multiprocessor is a set of processors, where each processor is connected in a hypercube topology; i.e., P and Pj are connected by a communication link if and only if the binary representations of i and j differ in exactly one bit. In a hypercube, there is no shared memory. The entire storage capability consists of constant size local memories, one attached to each processor.

2.2. Multi-Way Search On Trees And General Graphs

Before presenting our parallel inference algorithms, we introduce some notations and previous results on hypercube algorithms which will be used in the remainder. In particular, we use a procedure called multi-way search: Given a tree stored on a hypercube multiprocessor, m search queries on that tree are to be executed independently and in parallel. At each time step, each query currently visiting a node of the tree decides which adjacent node to visit next, and is then moved to that node. Note that, each node can be concurrently visited by an arbitrary number of queries.

More formally, let T = (V, E) be a tree of size k, height h, and out-degree O(1), and let U be a universe of possible query queries on T. A search path for a query q ∈ U is a sequence path(q) = v1, ..., vh of h vertices of T defined by a successor function f : (V \ {start}) × U → V (i.e., a function with the property that f(start, q) ∈ V and for every vertex vs V, (v.f(v, q)) ∈ E). A search process for a query q with search process path(v1, ..., vh) is a process divided into h time steps t1 < ... < th such that at time tj, 1 ≤ sj, query qj is matched with node vj. A match of a query q and a node vj at time tj is defined as a situation where there exists a processor which contains a description of both, the query q and the node vj. Note, however, that we do not assume that the search path is given in advance; we assume that it is constructed during the search by successive applications of the functions f. Given a set Q = {q1, ..., qm} ⊂ U of m queries, m = O(k), then the multi-way search problem consists of executing (in parallel) all m search processes induced by the m queries. In [3, 8] it was shown that the multi-way search problem can be solved on a hypercube multiprocessor of size max(k, m) in time O(log k log log k + h log k). It follows from [4, 7, 8] that for trees with unbounded out-degree, as well as arbitrary graphs, the time complexity increases to O(h log k log log k).

3 A Parallel Frame Based Knowledge Server Supporting Single Inheritance Hierarchies With Implicit Storage

In this section, we will study the efficient hypercube implementation of a knowledge base server supporting a single inheritance hierarchy. We first describe, in Section 3.1, how to store a frame based system on a hypercube multiprocessor, and then, how this representation can be effectively used for inference. We will be interested in answering two basic types of elementary queries: bottom-up and top-down inference queries [11]. These query are elementary and intended to serve as a base upon which more complex query types can be defined by higher level inference mechanisms (as depicted in Figure 2).

In the following Sections 3.2 and 3.3, we show how multiple bottom-up and top-down inference queries can be processed efficiently in parallel. To simplify exposition, we will describe our inference methods for both query types separately; it is however easy to see that both types of queries can be processed simultaneously.

3.1 Storing An Inheritance Hierarchy On A Hypercube Multiprocessor

We require a scheme for distributing an inheritance hierarchy over the local memories of a hypercube. Consider the level numbering of the frames of an inheritance hierarchy as indicated in Figure 4. For the remainder we will assume that each frame with level number i, together with its links and data, is stored at processor Pi.

In many systems (e.g., [17, 25]), inheritance is "precompiled" such that little or no searching is required to find the value of a slot, even if the value derived from some superclass via inheritance. While we will adapt such an approach for our parallel multi inheritance knowledge base server (Section 3), we will use an implicit representation for the single inheritance system to be discussed in this section. That is, in order to store a frame at a processor, we store only the explicitly valued slots. For implicitly valued slots, there is no reference necessary, since these values will be determined by the inference mechanism.

3.2 Answering Multiple Bottom-Up Inference Queries

We first consider the parallel implementation of multiple bottom-up queries. Each bottom-up query, q(X), is of the form "Does frame X meet condition a through b" or "What are the values of slots a through b of frame X?". "Is the Exxon-Valdez an vehicle with color black and current-location Alaska?" or "What is the weight of the QE2?" are examples of bottom-up queries based on Figure 3. Bottom-up queries are always about a particular instance or class frame but, since we
are using an implicit representation, may require the examination of all superclasses of that frame.

Queries will be represented by records of the form depicted in Figure 6. Each "Current Value" field of a query's slot is used to store the value of the respective slot at the frame the query is currently examining.

Consider an inheritance hierarchy with \( n \) frames, stored on a hypercube multiprocessor with \( N \) processors as described in Section 3.1 (w.l.o.g., \( n = N \)), and a set of \( m \) SN bottom-up inference queries where each query is stored at one arbitrary processor. For the remainder, \( frameset(i) \) and \( queryset(i) \) refer to the query and frame (currently) stored at processor \( PE(i) \).

Figure 7 outlines a hypercube algorithm for answering, in parallel, unlabeled top-down inference queries. It works essentially by first matching the queries to the frames they refer to and then shifting them through the tree towards the root until all slots referred to in the query have been instantiated. As indicated in Section 2.3, the problem of advancing all \( m \) queries one step along their path (in the inheritance hierarchy) towards root frame \( \text{root} \) can be solved on a hypercube multiprocessor of size \( n \) in time \( O(\log n) \) if the number of children of each frame is bounded by a small constant. For unbounded number of children, each advancement all \( m \) queries takes time \( O(\log n \log \log n) \) (4, 7, 8).

The remainder of Step 2 consists of simple local, \( O(1) \) time, operations. From [7, 8] and [4] it also follows that the initial match in Step 1 can be executed in time \( O(\log n \log \log n) \).

Summarizing, we obtain that all \( m \) unlabeled bottom-up inference queries can be answered, in parallel, in time \( O(\log n \log \log n) + O(\log \log n \log \log ^2 n) \) if frames can have an unbounded number of children, where \( n \) is the height of the inheritance hierarchy.

### 3.3 Answering Multiple Top-Down Inference Queries

Top-down inference queries, \( q(X) \), are of the general form "Identify all frames in the subtree (of the inheritance hierarchy) rooted at \( X \) such that conditions \( a \) through \( z \) are true". For example, "Identify all instances of Sea-Vehicles with weight \( > 1000 \) tons and paying passengers \( a \)", or "Identify all classes which are subclasses of Vehicle and have less-than 10 paying passengers" are top-down inference queries based on the hierarchy in Figure 3.

Again, queries are represented by records of the form depicted in Figure 6. Each "Current Value" field of a query's slot is used to store the (implicit) value of the respective slot at the frame the query is currently examining.

Figures 8 and 9 outline our hypercube algorithm for answering multiple top-down inference queries. Again, we assume an inheritance hierarchy of \( n \) frames, stored on a hypercube multiprocessor with \( N \) processors as described in Section 3.1, and a set of \( m \) SN top-down inference queries where each query is initially stored at one arbitrary processor. Figure 8 shows the general structure of the algorithm. Steps 1-3 are similar to our bottom-up inference algorithm. The result of these steps is that each query, \( q(X) \), has for all the slots which are specified in it, explicitly stored the implicit values at frame \( X \). What is left to do in the remaining steps is to search, for every \( q(X) \), the subtree rooted at \( X \). To this end, a search token and control token are created for every query. Each search token, for a query \( q(X) \), traverses (independently and in parallel) in preorder the subtree rooted at frame \( X \) and determines the answers to be reported; the details are described in Steps 5d and 5e, together with Figure 9. Each control tokens remains at the root of the respective subtree to be traversed, indicates to the respective search token the end of its traversal, and creates new parallel processes in the same way as search tokens do. (Note that, the number of control tokens never exceeds the number of search tokens.) The main idea leading to a near optimal speedup (as will be shown in Section 3.4) is to re-use processors released by queries which need to traverse smaller subtrees to improve the performance of the search processes for the larger subtrees. This rescheduling mechanism is described in Steps 5a-5c. After each "round", i.e. parallel advancement of all search tokens by one edge in the preorder of their subtree, processors from finished traversal processes are given to unfinished traversal processes. Every token determines, from the outdegree of the frame it is currently visiting, how many "assistants" it could currently utilize, i.e. ask them to search those subtrees independently and in parallel. For the distribution of available processors, the following heuristic is used: higher priority is given to those search tokens with smaller level number, i.e. tokens that have (in the expected case) the largest subtrees still to be searched.

For observing the correctness of the algorithm note that, the above algorithm searches for every query \( q(X) \) the entire subtree rooted at frame \( X \), and that at every time a frame is examined, all inherited values are present. Due to the rescheduling procedure, the performance analysis for this algorithm is more complicated than in Section 3.2 and will be discussed separately in the next section.

### 3.4 Analysis And Experimental Results

In the previous sections we introduced two inference algorithms. In the case of the bottom-up inference algorithm it was possible to get a worst case bound on the algorithm's time complexity. In the case of the top-down inference algorithm, worst case analysis is more difficult. As in Section 3.2, advancing all \( m \) tokens one step along their path can be executed on a hypercube multiprocessor of size \( n \) in time \( O(\log n) \) (if the number of children of each frame is bounded by a small constant) or in time \( O(\log n \log \log ^2 n) \) (for unbounded number of children); see Section 2.3. The problem lies in determining the number of such parallel steps required by our algorithm, since at the heart of the method is a heuristic rescheduling scheme that reallocates processors to queries. The challenge is to quantify how effective this reallocation technique is.

In order to test our mechanism, we have implemented a prototype system and have performed extensive tests using randomly generated hierarchies and sets of top-down inference queries. We considered the following input parameters:

- \( n \) = number of frames = number of processors,
- \( m \) = number of queries,
- \( k \) = max. number of children of a frame.

Figure 10 shows the result of our experiments for 16,000 node hierarchies (\( n = 16,000 \)). The graph on the left depicts results for hierarchies with unbounded \( k \), while the graph on the right shows results for hierarchies with a small value of \( k \) (\( k = 8 \)).

The x-axis in each diagram represents \( m \), the number of queries, ranging from 1 to 16,000 in 1% increments. For each value of \( m \), 1,000 experiments were performed, each with a new randomly generated hierarchy and set of queries. The two curves show the average number of parallel steps as well as the average speed up. The speed up was measured by comparing the number of parallel steps with the total number of steps necessary for sequentially processing the same query set on the same hierarchy. It measures the utilization of the massive parallel architecture and, as our results show, a nearly optimal utilization is obtained for a 75% load factor (number of processors, \( n \), divided by number of queries, \( m \)). The utilization never dropped below 75%, regardless of the load factor and other parameters.

The shape of the curves in Figure 10 can be explained by two opposing effects. If there are only a few queries to be processed (small load factor), these can not immediately
request enough assistants (due to the constraints of the hierarchy) to utilize all processors. On the other hand, it is important for large subtrees to receive assistants early in the traversal process. Hence, if the number of queries is close to the number of processors, there are no (or only very few) assistants available until the smaller trees have been traversed. Therefore, it becomes likely for the larger trees, that late arriving assistants can not be efficiently applied to the traversal.

### 3.5 Dynamic Knowledge Representation: Assert And Retract Queries

We have described how inference queries can be answered efficiently on a static frame based inheritance hierarchy of n nodes. We will now sketch how assert and retract queries can also be added to the system, thereby producing a truly dynamic knowledge representation scheme.

Assert queries are queries that add knowledge to our representation. There are three basic types of assertions: assertion of a new slot value, assertion of a new slot (with initial value), or lastly, assertion of a new class or instance frame (complete with slots and values). Retract queries fall into three analogous types: retraction of a slot value, retraction of a slot, and retraction of a class or instance frame.

Assert and retract queries are executed simultaneously to our search queries. From the semantic viewpoint, however, all assert and retract submitted in one round (set of inference/assertion/retraction queries to be processed in parallel) will be performed only after all inference queries in that round have been answered. In addition, assertion and retraction queries processed in parallel must be prioritized. (For example, it is possible that several assert queries may attempt to change the value stored in a particular slot and frame in the same round. Which if any of these changes should have a lasting effect?) We assume query priorities based on their position in the set of submitted queries, i.e., queries submitted towards the end of the list of queries (to be processed in parallel) are considered to be executed (logically) after those submitted earlier in the list.

Assert and retract queries can then be easily processed as follows: Match the queries with the respective frames, using the sorting algorithm in [4]; use the concentrate/distribute operations in [21] to remove redundant assert and retract queries; apply the dynamic multi-way search algorithm referred to in Section 3 to insert/delete the required frames. All of these operations can be computed, for all assert and retract queries in parallel, in time O(log n loglog^2 n).

### 4 Multiple Inheritance Knowledge Bases

With Explicit Storage

In this section we sketch how the above single inheritance system can be modified to obtain a multiple inheritance knowledge base. While we used an implicit storage scheme for the single inheritance system described above, we will apply an explicit storage mechanism for obtaining a hypercube implementation of a parallel multiple inheritance knowledge base server. That is, we assume that in every frame, each slot which is not explicitly valued contains pointers to all possible slots from whom it can inherit its value, together with a function computing from these values the actual inherited contents. [13, 17, 25]

With such an explicit storage system, bottom-up inference queries can be easily answered by matching the queries with the respective frames, and then matching them with the frames referred to by the pointers in the respective slots. Hence, all bottom-up inference queries can be answered in time O(log n loglog^2 n); see Section 2.3.

For each top-down inference queries q(X), the problem reduces to matching the query with the respective frames X, and then traversing the subtree of all frames who have X as their super class. Despite the fact that this subtree is not any more in level ordering, as for top-down inference queries in Section 5, the traversal is in Section 5 can essentially applied to this problem as well. The only difference is that for advancing all search tokens by one step in their preorder traversal, the sorting algorithm in [4] needs to be applied; this results in a time complexity of O(log n loglog^2 n) per parallel advancement instead of O(log n); see Section 2.3. Otherwise, the same analysis and experimental results as shown in Section 5.4 apply.

The execution of assert and retract queries becomes obviously more involved than in the implicit storage scheme. In addition to the update/insertion/deletion of frames described in Section 3.5, all possible pointers to those slots need also to be updated. We observe, though, that for each update/insertion/ deletion of frame X it is imperative to add all ancestors or at least all descendants of X, and update the pointers in those frames' slots. Hence, we obtain a parallel (multiple inheritance) assert/retract algorithm by adding to the assert/retract algorithm in Section 3.5 the same multiple subtree traversals as described in the previous paragraph. That is, again the same analysis and experimental results as shown in Section 3.4 apply.

### 5 Conclusion

In this paper we showed how to execute in parallel a set of inference and assert/retract queries on a shared knowledge base (with n frames) stored on a fine grained SIMD hypercube multiprocessor (with N=n processors). We studied single inheritance hierarchies and showed that m\textit{N} bottom-up inference queries can be answered, in parallel, in time O(log n loglog^2 n) [or O(log n loglog^3 n) if frames can have an unbounded number of children], where n is the height of the inheritance hierarchy. We presented a heuristic algorithm for answering multiple top-down inference queries: our experimental results showed that a nearly optimal (100%) processor utilization is obtained for a 70% load factor (number of processors divided by number of queries). We also outlined how assert and retract queries can be executed with essentially the same time complexity as top-down inference queries. We finally sketched how our system can be modified to manage multiple inheritance hierarchies with explicit storage.

### Acknowledgements

We acknowledge the assistance of Alain Bonopera, Kevin Bourgault, Vincent Ho, Joel Luculc, and Desmond Weong in implementing the prototype and multi-way search.

### References


---

Figure 1. Many Users Sharing a Single Large Knowledge Base Server.

Figure 2. A Layered View of a Knowledge-Base Application.

Figure 3. An Example of an Inheritance Hierarchy.
Algorithm 2: Answering Multiple Top-Down Inference Queries

1) Match each query q(X) with the frame X it refers to.
2) As long as there is still a processor PE(i) storing a query with Completed = False, repeat the following:
   2a) Every PE(i): If any slot in frame(i) is explicitly valued and query(i) refers to the same slot but is currently unvalued, set the value of that slot of query(i) to the value given in the frame.
   2b) Every PE(i): If all necessary slots referred to in query(i) have been instantiated, check the condition and set Completed to True.
   2c) Use multi-way search to advance all query(i) with Completed = False one step along their path (in the inheritance hierarchy) towards root, i.e., match them with the ancestors of the frame currently examined.
3) Match each query q(X) with the frame X it refers to.
4) Split each query q(X) into two tokens, a search token search token and a control token control token. Each token contains a copy of the original query. Each search token at frame X is responsible for searching the subtree rooted at the first child of frame X; each control token at frame X is responsible for searching (or having searched) the subtrees rooted at the other children of frame X.
5) As long as there is still a processor PE(i) storing an unanswered query q(X), repeat the following:
   5a) Count the number F of free processors. A free processor is a processor that is currently not supporting any search token.
   5b) Each token t calculates the number a(t) of assistants it could currently use. A control token can always use as many assistants as it has remaining subten to search. A search token can always use as many assistants as there are unsearched subten at the frame it is currently visiting. F new search tokens (assistants) are created and matched with the existing (search and control) tokens in order of the level numbering of their frames, each receiving a(t) assistants until all new tokens are distributed.
   5c) For each token that has been allocated assistants in Step 4, assign a child whose subtree has not been searched yet to each assistant, match the assistants with those children, create for each a corresponding control token, and have them search the respective subtrees.
   5d) Execute "Process-Search-Tokens" as shown in Figure 9.
   5e) Use multi-way search to advance all search tokens.

Algorithm 1: Answering Multiple Bottom-Up Inference Queries

1) Match each query q(X) with the frame X it refers to.
2) As long as there is still a processor PE(i) storing an unanswered query(i), repeat the following:
   2a) Every PE(i): If any slot in frame(i) is explicitly valued and query(i) refers to the same slot but is currently unvalued, set the value of that slot of query(i) to the value given in the frame.
   2b) Every PE(i): If all necessary slots referred to in query(i) have been instantiated, check the condition and set Completed to True.
   2c) Use multi-way search to advance all query(i) with Completed = False one step along their path (in the inheritance hierarchy) towards root, i.e., match them with the parent of the frame currently examined.

Figure 4. Level Numbering of the Nodes of a Single Inheritance Hierarchy

Figure 5. A Frame record representing the class Oil-Tankers

Figure 6. A query record representing the bottom-up query "Identify all instances of Sea-Vehicle with weight > 1000 tons and paying passengers = 0"

Figure 8. Hypercube Algorithm for Answering Multiple Top-Down Inference Queries
Procedure "Process-All-Search-Tokens"

5a) For every slot \( j \) in frame(i) which is explicitly valued,
    if query(i) refers to the same slot then
        if the last node visited by the search token
            was the parent of frame(i)
            then the inherited value for slot \( j \) of frame(i) is
                the current value of slot \( j \) of the search
token; the explicit value of slot \( j \) (if any)
of frame(i) becomes the current value of
slot \( j \) of the search token; check the query
condition and report the result (if
condition = true).
else the explicit value of slot \( j \) (if any) of
frame(i) becomes the current value of slot
\( j \) of the search token.

5b) If the search token has not yet arrived at its
    control token
    then select, as frame to be visited next, the next
    node in the preorder traversal.
    else if the search token's corresponding control
token has additional subtrees to be
searched
    then start traversing one of those subtrees.
    else delete both, the search token and the
    corresponding control token, and release
    the processor.

Figure 9. Processing of Search Tokens

Figure 10. Experimental results for Top-Down inference