Enhancing the Performance of OpenLDAP Directory Server with Multiple Caching

Jong Hyuk Choi and Hubertus Franke
IBM Thomas J. Watson Research Center
P.O. Box 218, Yorktown Heights, NY 10598
{jongchoi, frankeh}@us.ibm.com

Kurt Zeilenga
The OpenLDAP Foundation
kurt@OpenLDAP.org

Keywords—Directory, LDAP, Caching, Linux, Profiling.

Abstract—Directory is a specialized data store optimized for efficient information retrieval which has standard information model, naming scheme, and access protocol for interoperability over network. It stores critical data such as user, resource, and policy information in the enterprise computing environment. This paper presents a performance driven design of a transactional backend of the OpenLDAP open-source directory server which provides improved reliability with high performance. Based on a detailed system-level profiling of OpenLDAP on the Linux OS, we identify major performance bottlenecks. After investigating the characteristics of each bottleneck, we propose a set of caching techniques in order to eliminate them: directory entry cache, search candidate cache, IDL (ID List) cache, IDL stack slab cache, and BER (Basic Encoding Rule) transfer cache. The performance evaluation with a directory workload convinces that these caches, when combined, yields a 126% throughput increase and a 59% latency reduction with a reasonable level of storage overhead.

I. Introduction

A directory provides a logically centralized view of information in a distributed environment, enabling different platforms to access a shared, consistent information base. By sharing critical information such as user, resource, and policy data, interoperability among heterogeneous systems and services can be significantly enhanced. As IT services become available to customers in a dynamic, on-demand basis, it becomes vital to provide standard ways of information access and sharing.

LDAP (Lightweight Directory Access Protocol) is a standard access protocol for OSI X.500 [1] conforming directories. Because LDAP can run on top of TCP/IP instead of the OSI protocol stack, LDAP was first used to alleviate the client side protocol overhead. LDAP clients connect to an LDAP gateway that forwards requests and responses to and from an X.500 directory. Further in the direction of being lightweight, it has become commonplace to use standalone LDAP directory servers that store directory data directly in them without the need of separate X.500 directories.

Because directory searches constitute the majority of directory operations, it is particularly important to provide a high performance directory search in terms of latency and throughput. Low latency is essential to low delay IT services that rely on the directory. High throughput is essential because one directory server should be able to process a large number of requests from multiple directory clients simultaneously. It is also important to provide a highly reliable directory service since critical data are usually stored in enterprise directories.

This paper presents our efforts to enhance OpenLDAP [2], an open-source directory software suite. More specifically, this paper focuses on the performance and reliability of slapd, the standalone LDAP server of OpenLDAP. We participated in the development of back-bdb, a transactional backend of OpenLDAP, which directly utilizes the underlying Berkeley DB without the general backend API layer used in the existing OpenLDAP backend, back-ldbm. Back-bdb provides a higher level of reliability and concurrency. The transaction support of back-bdb makes directory recoverable from temporary failures and disasters, while the page level locking enables concurrent access to directory by the directory server and various administrative tools at the same time. However, because the initial design of back-bdb did not exhibit the expected performance, we analyzed its performance through a detailed system-level profiling. Based on the bottleneck identification and analysis, we propose five distinct caches for OpenLDAP back-bdb: entry cache, candidate cache, IDL (ID List) cache, slab cache for the IDL stack, and BER (Basic Encoding Rule) [3] cache for transfer contents. The combined use of these caches yields a performance improvement of 126%. This paper analyzes the efficacy and the performance impact of these caches in detail.

The next section will introduce LDAP and the OpenLDAP open-source project. Section III will describe the architecture of OpenLDAP directory server, slapd, focusing on the design of the search operation in back-bdb. Section IV will introduce five caching approaches proposed in this paper. Section V will describe the experimental setup used throughout the paper. In Section VI, the design and performance analysis of the entry cache for back-bdb will be described. After presenting the profiling mechanism used to identify performance bottlenecks in Section VII, the following three sections present the design and performance analysis of the four caches : the candidate cache in Section VIII, the IDL cache and the IDL stack slab cache in Section IX, and the BER cache in Section X. Section XI concludes the paper.

II. LDAP and OpenLDAP

LDAP (Lightweight Directory Access Protocol) [4] is a standard directory access protocol of the Internet to access directories having the X.500 [1] naming and data models [5].

LDAP defines an access protocol over TCP/IP that is a well defined subset of the X.500 DAP (Directory Access Protocol) to enable lightweight implementations. Its protocol syntax is defined in ASN.1 (Abstract Syntax Notation One) [6]. LDAP provides ten directory operations : search, compare, add, delete, modify, modifyDN, bind, unbind, abandon, and extended. The exchanged protocol messages between the server and the client are encoded by using BER of ASN.1. LDAP uses a restricted form of BER to reduce the complexity of the BER encoding/decoding process [4]. The attribute values are in a string format (`LDAPString`) in LDAP.

LDAP uses a subset of the X.500 naming and data models and organizes directory entries hierarchically in a DIT (Directory Information Tree). A DIT can be composed of multiple subtrees residing on different LDAP servers. A referral mechanism is provided to allow clients to chase a link across servers when they encounter such boundaries. On the other hand, an LDAP server may host a forest consisting of multiple DITs.

An entry is identified by a unique name called RDN (Relative Distinguished Name) among its siblings under the common superior entry and by a unique name called DN (Distinguished Name) within the entire DIT. One or more attribute values of an entry form RDN of the entry. DN can be formed by concatenating RDN and DN of its immediate superior.

An entry consists of a set of attributes permitted by the entry's object class defined in the directory schema. and system and user schema definitions. An attribute consists of an attribute type followed by one or more attribute values. The attribute type designates the name and OID (object identifier) of the attribute and the syntax, the matching rules, and the cardinality of the attribute values [7]. The object class designates entry's name, OID, description, and its superclass as well as the required and allowed attributes. The object class hierarchy represents the class hierarchy of entries, whereas DIT represents the hierarchy of directory entry objects. An object class inherits attributes from its superclass.

OpenLDAP [2] is an open-source directory software suite conforming to the LDAPv3 protocol [4] and supporting various platforms including Linux, FreeBSD, Apple Mac OS/X, Sun Solaris, Microsoft Windows. Currently, it sports a rich set of features [8] : LDAPv3 over both IPv4 and IPv6, SASL (Simple Authentication and Security Layer) support, TLS (Transport Layer Security) / SSL (Secure Socket Layer) support, access control, internationalization, multiple database instances, multi-threading, replication, configurability, generic modules API for extension, and various backends.

From OpenLDAP 2.1, back-bdb, a transactional backend which directly interacts with the Berkeley DB, is being provided as a default backend. Because the database access is transaction-protected and write ahead logs are maintained while database is modified, directory operations can roll back from a temporary failure and directory data can be recovered from a database crash. In addition, multiple directory access and modification requests can run with high concurrency which leads to high performance operations. Also it is possible to run multiple applications that access a single directory database at the same time.

The OpenLDAP software suite consists of a standalone directory server (slapd), a replication daemon (slurpd), client APIs (C, C++, Perl ...), and various client and server side tools. Slapd is a multi-threaded directory server that can be easily configured to support various types of backends.

The OpenLDAP directory software suite is currently being deployed as the default directory software in most Linux distributions including RedHat and SuSE. OpenLDAP is also being widely used in many enterprise IT environments.

III. LDAP SEARCH OPERATION

This section will introduce the search operation and the architecture of OpenLDAP slapd directory server.

A. LDAP Search Request

The following is the definition of the search request [4]:

```
SearchRequest ::= SEQUENCE {
  baseObject   LDAPDN,
  scope        ENUMERATED {
         baseObject              (0),
         singleLevel             (1),
         wholeSubtree            (2) },
  derefAliases ENUMERATED {
         neverDerefAliases       (0),
         derefInSearching        (1),
         derefFindingBaseObj     (2),
         derefAlways             (3) },
  sizeLimit    INTEGER (0 .. maxInt),
  timeLimit    INTEGER (0 .. maxInt),
  typesOnly    BOOLEAN,
  filter       Filter,
  attributes   AttributeDescriptionList }
```

`baseObject` defines the DN of the base object entry, the reference point relative to which the search is performed. If `scope` is `baseObject`, only the `baseObject` is searched; if `singleLevel`, all direct subordinate entries of the base object are searched; if `wholeSubtree`, the entire subtree rooted at the base object is searched. `derefAliases` indicates whether to dereference an aliases encountered during the search. Possible options are 1) always-dereference, 2) never-dereference, 3) dereference only upon positioning the base object, and 4) dereference except for the base object positioning. `sizeLimit` restricts the number of entries to be transmitted as the result of a search, while `timeLimit` restricts the maximum time in seconds allowed for a search. If `typesOnly` is set to TRUE, only the attribute types will be sent to the client without the attribute values. `filter` defines the matching condition of the search. The basic component of the filter is `AttributeValueAssertion` (AVA) that tests an attribute against a value according to a matching rule. `attributes` specifies a list of attributes to be transmitted to the client when an entry matches the search filter. An empty list or "*" in `attributes` means that all user attributes are requested.

B. OpenLDAP Search Implementation

Fig. 1 illustrates how the search operation is performed by the slapd OpenLDAP directory server. Also shown in the

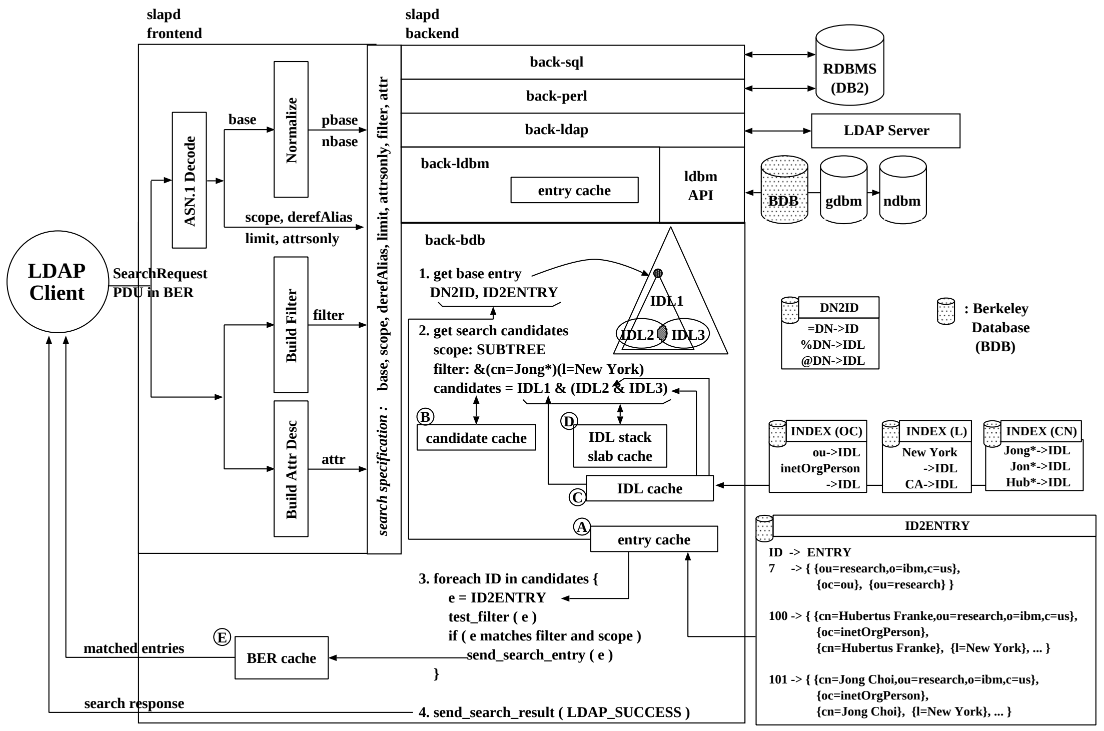


Fig. 1. Architecture of the OpenLDAP Directory Server (slapd) : Search Operation.

figure is the architecture of the OpenLDAP directory server viewed from the perspective of the search operation.

Slapd consists of the frontend and the backend. The frontend manages the LDAP connections with clients and manages the pool of LDAP worker threads. When a request arrives from a client, the frontend dispatches an available worker thread from the thread pool to process the request. Then, as shown in Fig. 1, it extracts the search specification from the LDAP client's search request PDU (protocol data unit) and build the internal representation of the search specification : the base object DN is normalized, the internal representation of the search filter is formed, and the list of attribute descriptions is built from the attribute names requested by the client.

Fig. 1 also illustrates the implementation of the search operation inside the back-bdb backend. Back-bdb maintains three distinct types of databases : DN2ID, INDEX, and ID2ENTRY. The Sleepycat BDB stores in the database (key, data) pairs. The data can be searched for by using the corresponding key value as the index. In back-bdb, every directory entry is given a unique identifier, ID. The ID2ENTRY database stores (ID, directory entry) pairs in it. A directory entry can be retrieved by using the corresponding ID as the access key.

The DN2ID database stores (xDN, IDL) pairs, where $x = \{=, \%, @\}$. DN is the name of the search base. IDL is an ID or a list of IDs of either the base entry ($=$), immediate subordinates ($\%$), or the whole subtree under the base entry ($@$).

The INDEX database is created for fast, indexed searches. The INDEX database stores (key, IDL) pairs, where key is the value of an attribute possibly used for matching. (key, IDL) pairs for all possible key values under the specified indexing methods are stored in the INDEX database corresponding to the attribute.

Back-bdb first retrieves the base entry from the database. As the DN of the base is given, we first retrieve the ID of the base from the DN2ID database. Then, we retrieve the entry from the ID2ENTRY by using the retrieved ID as the access key.

Back-bdb, then, builds the IDL of the search candidate entries. The building of the search candidate IDL is heavily relying on the indexing. Not all entries in the search candidates may turn out to be the matching entry of the search because of combined or inappropriate indexing.

The next step is to loop through the IDs in the search candidate IDL to select directory entries that actually match the filter. Because the attribute types and values of an entry can be accessed after the entry is retrieved from the ID2ENTRY database in the IDL loop, we can test the scope and the filter with the entry. If the entry matches, back-bdb transmits the entry to the client after encoding the entry by using the BER encoding of ASN.1. After iterating all the candidate entries in the IDL, back-bdb notifies the client of success or failure by sending the search response.

IV. Proposed Caching Approaches

In order to improve the performance of back-bdb, we propose the following caching mechanisms in this paper.

The first is an entry cache (labeled A in Fig. 1) for back-bdb. As in back-ldbm, the entry cache of back-bdb stores in memory the directory entries recently retrieved from the ID2ENTRY database. The cache is organized as two AVL trees, one keyed by the entry DN and another by the entry ID. The DN AVL tree is used for the base object access, while the ID AVL tree is used for the candidate entry access. The entry cache is integrated into the slapd back-bdb in a way that cooperates with the database transaction rollback and locking mechanisms to ensure consistency and deadlock freedom.

The second one is a candidate cache (labeled B in Fig. 1) that stores the recently established search candidate IDLs. Through a detailed performance analysis, we confirmed that it is very effective in improving the performance of the search-only directory access scenario. The two performance bottlenecks of back-bdb identified by system-level profiling, the index database access and the IDL stack management overhead, proved to be successfully eliminated by the candidate caching.

Although the search-only directory is not uncommon, it is more beneficial to provide a general solution for both the search-only and the search-update workloads. In order to reduce the index database access overhead, we propose an IDL cache (labeled C in Fig. 1) that stores in memory the recently accessed IDLs from the INDEX and the DN2ID databases. In order to reduce the IDL stack allocation and deallocation overhead, we propose a slab cache (labeled D in Fig. 1) that retains once allocated memory chunks of the IDL stack. The combined use of the IDL and the IDL stack slab cache proved to effectively improve the performance of both the workloads.

We also propose a BER cache (labeled E in Fig. 1) that stores the BER representation of the search result entries. A preliminary performance result shows a sizable performance gain.

V. Experimental Frame

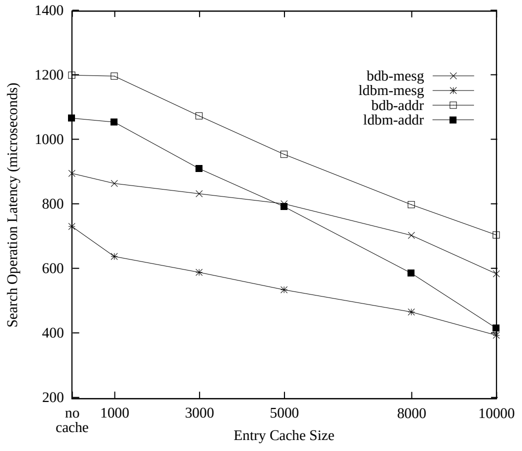
The performance of directory servers was measured by using the DirectoryMark [9] benchmark suite which provides a set of tools to generate LDIF files and test scripts along with the request generation engine. The generated DIT is an enterprise directory consisting of 10 organizational units. The `inetOrgPerson` entries are regimented under the organizational units. Two different types of LDAP use scenarios were used as input workloads in the experiment: 1) The messaging scenario models a messaging server (i.e. mail transfer agent) searching for routing information by contacting a directory server. All search requests are exact matches on the `UID` attribute of `inetOrgPerson`. It is assumed to bind only once at the start of the run. 2) The addressing scenario models the behavior of address lookup clients searching for user information. The requests consist of a sequence of searches with a predefined filter combinations: 24% of them are substring matches for the `UID` attribute and 8% of them are not-found; the rests are the equality matches for a small number of attributes. It is assumed that there is one anonymous bind every five requests.

A 10,000-entry DIT was used for the test. Attributes returned to the clients are `objectClass`, `cn`, `sn`, `description`, `facsimileTelephoneNumber`, `l`, `postalAddress`, `telephoneNumber`, and `title`.

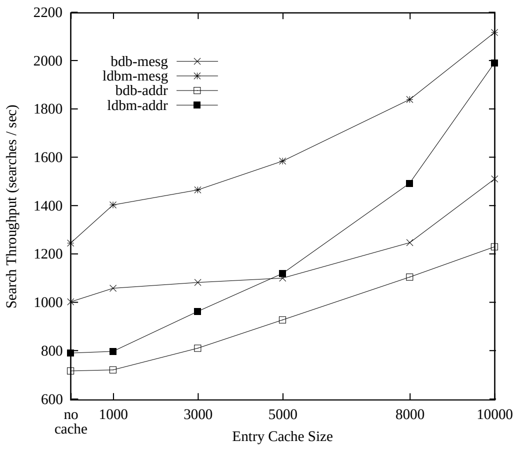
The directory server system under test is an IBM Intellistation (Pentium III 1GHz, 512MB of main memory) which runs RedHat Linux 7.3 (Kernel ver 2.4.18-3). The DirectoryMark client is configured to consist of 16 request threads.

VI. Entry Cache for Transactional Backend

The back-bdb backend of OpenLDAP 2.1 utilizes the transactional data store of BDB in order for multiple threads to access databases concurrently. A page level read-write locking is implemented by BDB for concurrent access. BDB avoids deadlocks by returning a deadlock error. With the help of the transactional support, applications can abort a transaction upon a temporary error situation such as deadlock error and roll back



(a) Latency



(b) Throughput

Fig. 2. Performance Comparison of back-ldbm and back-bdb.

TABLE I. Linux Execution Profile of OpenLDAP 2.1 Server with back-bdb.

slapd	31.47%	**libdb-4.0**	26.99%	**vmlinux**	13.79%
ber_printf	2.01%	__ham_item	6.55%	do_zap_page_range	2.26%
send_search_entry	1.81%	__ham_lookup	6.54%	get_unmapped_area	0.72%
ber_write	1.51%	__ham_get_cpage	2.22%	save_i387	0.60%
is_ad_subtype	1.48%	__ham_item_next	1.81%	do_signal	0.39%
ad_inlist	1.29%	__lock_get_internal	1.16%	schedule	0.38%
libc	15.57%	**libpthread**	9.09%	**misc**	3.09%

TABLE II. Linux Execution Profile of OpenLDAP 2.1 Server with back-ldbm.

slapd	42.72%	**libdb-4.0**	4.44%	**vmlinux**	11.44%
ber_printf	2.58%	__bam_cmp	0.87%	save_i387	0.77%
send_search_entry	2.46%	__bam_search	0.61%	schedule	0.70%
ber_write	2.05%	__memp_fget	0.40%	send_sig_info	0.56%
is_ad_subtype	1.99%	__db_c_get	0.29%	restore_i387	0.54%
ber_put_seqorset	1.74%	__memp_fput	0.27%	tcp_sendmsg	0.53%
libc	23.08%	**libpthread**	14.06%	**misc**	4.26%

from the start without committing any changes to the database. On the next try of the same database access, chances are high that the cause of the error has disappeared. Upon a permanent error or crash, on the other hand, applications can recover from the write ahead logs to restart from a known, clean state.

The improved reliability should be achieved without compromising performance - especially for searches, because directories usually store mostly-read data. One of the most effective mechanism to boost the directory search performance is to cache recently accessed directory entries in the memory [10].

However, it is not straightforward to design a directory entry cache for back-bdb, because it has its own locking and transaction mechanism for concurrent database access The entry cache of back-ldbm has a relatively straightforward design because there is only one giant lock for backend access.

Because a database page lock will not be released before transaction commit and multiple database objects can reside in a single page, deadlock situations may arise if the entry cache starts using its own locking mechanism. Through discussions in the OpenLDAP community, we have decided to utilize the locking mechanism of BDB for the entry cache as well. Hence, the access to cache entries was made to be protected by the deadlock avoidance mechanism of the BDB page locking.

The entry cache also interferes with the transaction because it does not have any rollback capability and it is possible for a cache update to fail. We've utilized the two phase commit mechanism of BDB. Before updating the entry cache, `TXN_PREPARE()` is invoked to ensure that the transaction is guaranteed to commit. If so, the cache is updated and in turn the transaction is committed by `TXN_COMMIT()`. Otherwise, the transaction is aborted and rolled back to start without making any changes to the entry cache.

Fig. 2 illustrates the performance of OpenLDAP 2.1 server with back-ldbm and back-bdb backends. Fig. 2 (a) shows the search operation latency (`do_search()`) in μsec and Fig. 2 (b) shows the search throughput in operations per second. The latency and throughput of the messaging and address look-up scenarios are shown together. The performance of the address look-up scenario is lower than that of the messaging scenario, and more sensitive to the changes in entry cache size. This is because 24% of search requests are substring searches resulting in multiple search candidates. Because every candidate entry should be retrieved from the database to check for a match and should be transmitted if it matches, the address look-up scenario requires more computing, IO, and network bandwidth per search request than does the messaging scenario. The effect of cache misses is more significant when there are more entries to check for a search request.

Although the entry cache for back-bdb improved throughput by 50.5% (1006 to 1514) for the messaging scenario and 71.2% (721 to 1234) for the addressing scenario, the performance of back-bdb turned out to be lower than that of back-ldbm.

VII. PROFILING OF OPENLDAP UNDER LINUX

In order to gain insights on potential performance bottlenecks, we performed a system-level profiling of OpenLDAP 2.1 server running on top of Linux. We used `tprof`, IBM Trace Facility for Linux (x86), which can collect execution profiles of all execution entities of the Linux OS including user processes (thread-level), libraries, and the kernel itself. It is ideal for OpenLDAP 2.1 performance analysis, because the execution profile can be obtained without any loss of concurrency and different sub-modules of the OpenLDAP directory server, i.e. the slapd code, BDB and system libraries, and Linux kernel routines, can be analyzed in detail down to the function level. Tprof provides the number of performance counter events of each routine that was running when a predefined event occurs. We used `INST_RETIRED` performance counter event [11] of Pentium with 10000x sampling.

Table I and Table II show profiling results of slapd with back-bdb and back-ldbm backends, respectively. Symbols in boldface represent different entities of the OpenLDAP directory server and the Linux OS: slapd threads, the Linux kernel (vmlinux), the BDB library (libdb), the C library (libc), the pthread library (libpthread), and miscellaneous entities. Top

TABLE III. Linux Execution Profile of OpenLDAP 2.1 Server with Candidates Cache.

slapd	44.14%	**libdb-4.0**	3.45%	**vmlinux**	12.69%
send_search_entry	2.70%	__lock_get_internal	0.49%	save_i387	0.87%
ber_printf	2.69%	__ham_func5	0.36%	schedule	0.69%
avl_find	2.27%	__ham_lookup	0.36%	send_sig_info	0.64%
ber_write	2.24%	__ham_item	0.23%	restore_i387	0.61%
is_ad_subtype	2.18%	__lock_put_internal	0.21%	do_signal	0.54%
libc	21.54%	**libpthread**	13.66%	**misc**	4.52%

five most frequently observed routines are shown for slapd, libdb, and vmlinux. From the comparison of Table I and Table II, we can see that 1) slapd of back-bdb is less frequently observed than that of back-ldbm upon performance counter events; 2) libdb of back-bdb occupies much larger proportion of the execution and much of them are attributed to ham (hash access method) routines; 3) `do_zap_page_range()` and `get_unmapped_area()` routines of the Linux kernel are frequently observed with back-bdb, but not with back-ldbm.

Because the hash access method is used only for the INDEX database of back-bdb design (other databases make use of the btree access method), the indexing database access turns out to be the source of contention. The indexing database contention is twofold: 1) back-bdb stores individual ID in an IDL as a separate (key, data) pair by using the cursor operation with `DB_MULTIPLE` option; 2) there is an overhead of the transaction environment even for non-transactional operations.

`get_unmapped_area()` is a Linux kernel routine that finds an unused virtual memory range for memory allocation, while `do_zap_page_range()` removes user allocated page frames that map to a given virtual memory range [12]. Frequent invocations of these routines mean that memory is allocated, used, and deallocated frequently. What makes situations worse is that `do_zap_page_range()` requires page table operations, a flush of cache contents (in some architectures), and a TLB shootdown. The IDL stack allocation and deallocation in the `search_candidates()` was identified as the source of these memory mapping / unmapping kernel calls. An IDL stack is $((\texttt{depth}+1)\times \texttt{BDB\_IDL\_UM\_SIZE} \times sizeof(\texttt{ID}))$ in size. Because the minimum `depth` value is 2 and `BDB_IDL_UM_SIZE` is 128K, IDL stack size is at least 1.5MB. The memory allocation and free routines (`malloc()`/`free()`) uses `mmap()` instead of `brk()` for objects larger than a threshold.

VIII. Search Candidates Caching

Because both the sources of the back-bdb contention are attributed to the search candidate list construction, we first designed and implemented a candidate cache that stores established search candidates in an AVL tree. Search candidates in `bdb_search()` are represented as an IDL that is obtained by combining (union or intersection) multiple IDLs from DN2ID and/or INDEX databases. The candidate cache is indexed by the base, scope, and filter of the search request. The candidate cache is implemented as a separate AVL tree for each base entry. An optimization of the candidate caching is that we can remove unmatched entries from the candidate cache entry in memory as we iterate the candidate entries to find actual matches in order to make candidate list more accurate.

For the messaging scenario, back-bdb with the candidate cache exhibits 349.5 μsec of the search latency and 2378 searches/sec of search throughput. Compared to the back-ldbm's performance shown in section VI, back-bdb equipped with a candidate cache performs better. In the experiment, the candidate cache is constructed at the context prefix because it is the search base of the messaging scenario. The size of the candidate cache was set to unlimited.

Table III shows the profiling result of back-bdb with the candidate cache. We can see that the utilization of the slapd code is increased from 31.47% to 44.14%. This is in fact higher than that of back-ldbm (42.72%). The number of libdb invocations dropped significantly from 26.99% to 3.45%. The candidate cache proved very effective in reducing the pressure to the hash access method. From the table we can also observe that the page allocation and deallocation routines of vmlinux do not show up with the introduction of the candidate caching.

Although the candidate cache is very effective for the search performance improvement, the advantages started diminishing with the increasing presence of the update requests. This is because the entire candidate cache entries are to be effectively invalidated once a single update operation is processed. Because search candidates are constructed entities from multiple IDLs, it is not easy to invalidate only those candidate cache entries that should be invalidated when there is a change in the INDEX or DN2ID databases. With a timestamp-based lazy invalidation scheme, the performance of back-bdb with the candidate cache proved unaffected by the presence of update requests at one entry addition per 1000 messaging scenario searches. As update frequency increases, however, the candidate caching would have diminishing returns. Another disadvantage of the candidate caching is the unbounded size of candidate cache for 100% of coverage of search requests. A new candidate cache entry will be created whenever a search request having a new search specification arrives.

IX. IDL and IDL Stack Caching

Two major disadvantages of the candidate caching, the invalidation and the unbounded size problem, can be solved by having a cache that directly stores the INDEX and DN2ID database entries, i.e. IDLs. The IDL caching is implemented inside the `bdb_idl_fetch_key()` routine as an AVL tree. Upon insertion or deletion of an ID in the databases, only the affected index database entry is invalidated (in `bdb_idl_insert_key()` and `bdb_idl_delete_key()`). Also the size of the IDL cache is bound by the size of the INDEX and DN2ID databases. The IDL cache is keyed both by the identity of the database and the key of the database entry.

The dotted lines in Fig. 3 show (a) latency and (b) throughput of back-bdb with the IDL cache for the messaging scenario. X-axis is the size of the IDL cache in the number of entries. The left-most points are the latency and the throughput of the back-bdb without IDL caching. An LRU replacement scheme is used when an overflow occurs in the IDL cache.

The IDL cache in the experiment proved to reduce latency by 17.4% from 586.6 μsec to 484.54 μsec and to boost throughput by 17.8% from 1514 ops/sec to 1784 ops/sec. The initial performance degradation observed at the IDL size of 1000 entries is from the overhead of IDL cache management.

Table IV shows the execution profile of back-bdb equipped with the IDL cache. The utilization of slapd is 40.37%. We can see a significant reduction of libdb hash access method invocations due to the introduction of the IDL cache.

We identified the IDL stack management as the source of frequent page allocation / deallocation in the kernel. The IDL stack should be allocated and deallocated for every search operation because the size of the IDL stack is larger than the `mmap()` / `brk()` threshold as explained in Section V.

In order to eliminate frequent allocation and deallocation, a slab cache is designed for the IDL stack. Because the IDL stack size is determined by `depth`, we provide slabs of different sizes according to `depth`. Because its value is small in common cases, it is usually not required to provide large slabs. The maximum number of slabs that can be in use at the same time is limited by the number of slapd worker threads.

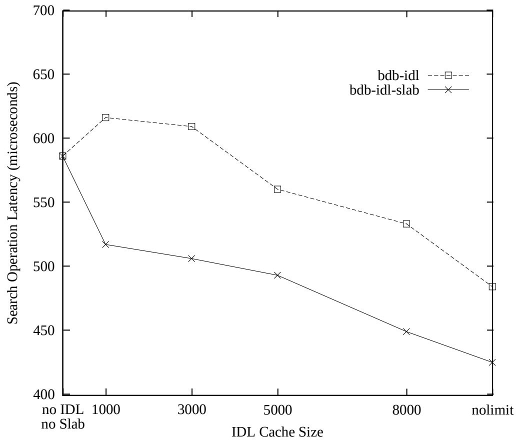
Table V shows an execution profile of back-bdb equipped with both the IDL cache and the IDL stack slab cache. As expected, the invocation of `do_zap_page_range()` and `get_unmapped_area()` do not show up in the execution profile after adding the IDL stack slab cache. The slapd utilization is 43.31% which is slightly higher than that of back-ldbm.

The solid lines of Fig. 3 show the search (a) latency and (b) throughput of back-bdb equipped with both caches. The steep decrease in latency and increase in throughput from no IDL / no Slab to 1000-entry IDL / Slab proves the effectiveness of the IDL stack slab cache. The performance boost between these two points is around 12%. Considering the performance overhead observed by the introduction of the IDL cache, the actual performance benefits by the slab cache must be larger. The improvement by the full IDL caching with the IDL stack slab caching is 27.5% reduction in latency (from 587 to 426) and 31.6% increase in throughput (from 1514 to 1993).

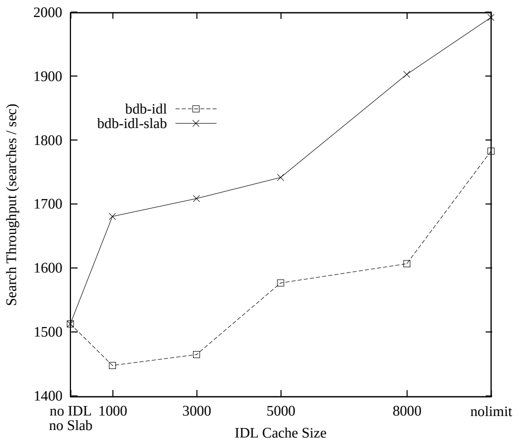
X. BER Caching of Transfer Contents

We also propose a BER cache that stores transfer content between the LDAP server and client. Because a large portion (more than 10%) of the execution profile events belongs to BER encoding routines such as `ber_printf()` and `ber_write()`, caching BER encoded representation of searched entries should improve performance. Because a BER cache entry represents an incarnation of the entry with a specific set of attribute types and values, which are subject to the search filter, the attribute set, and the access control lists, the BER cache will be effective in rather static directory use scenarios where a set of search specifications are pre-determined and seldom change. Example is the Bluepage directory of IBM. The Bluepage web application has a clearly defined set of pre-determined search filters and attribute sets. Access control information does not change frequently in such a directory.

A prototype BER cache design proves its effectiveness with the DirectoryMark messaging scenario: a 7.1% throughput increase is obtained when we add a BER cache to back-bdb with no IDL / no Slab cache, while a 13.3% throughput increase is



(a) Latency



(b) Throughput

Fig. 3. Performance of back-bdb with IDL/Slab Cache.

TABLE IV. Linux Execution Profile of OpenLDAP 2.1 Server with IDL Cache.

slapd	40.37%	**libdb-4.0**	7.14%	**vmlinux**	16.64%
ber_printf	2.28%	__lock_get_internal	0.69%	do_zap_page_range	2.82%
send_search_entry	2.24%	__ham_lookup	0.67%	save_i387	0.76%
is_ad_subtype	1.96%	__db_tas_mutex_lock	0.46%	schedule	0.68%
ber_write	1.92%	__ham_item	0.45%	send_sig_info	0.61%
avl_find	1.86%	__ham_get_cpage	0.39%	get_unmapped_area	0.50%
libc	19.33%	**libpthread**	12.78%	**misc**	3.74%

TABLE V. Linux Execution Profile of OpenLDAP 2.1 Server with IDL / Slab Cache.

slapd	43.31%	**libdb-4.0**	7.41%	**vmlinux**	11.43%
send_search_entry	2.51%	__lock_get_internal	0.75%	save_i387	0.85%
ber_printf	2.39%	__ham_lookup	0.66%	schedule	0.74%
is_ad_subtype	2.00%	__ham_item	0.47%	send_sig_info	0.59%
avl_find	1.98%	__ham_get_cpage	0.41%	restore_i387	0.52%
ber_write	1.91%	__db_tas_mutex_lock	0.38%	tcp_sendmsg	0.48%
libc	20.43%	**libpthread**	13.79%	**misc**	3.63%

obtained when we use a BER cache together with IDL and IDL stack slab caches. From the execution profile, we can see that only 3.42% of execution events belongs to the BER encoding routines after the addition of the BER cache.

XI. Conclusions and Future Works

In this paper, we analyzed the performance characteristics of OpenLDAP 2.1 directory server, identified performance bottlenecks, and provided appropriate solutions. First, we designed an entry cache for back-bdb, a transactional backend for OpenLDAP 2.1. Although the entry cache succeeded to improve back-bdb's performance significantly, back-bdb's performance turned out to be relatively low. By using a detailed system profiling, we identified the index database access and the IDL stack management as the two major sources of the back-bdb's performance degradation. We first proposed the search candidate cache to eliminate these bottlenecks as a proof of concept design. We, then, proposed the IDL cache to reduce the index database access cost and the IDL stack slab cache to eliminate frequent allocation and deallocation of large chunks of memory. We verified the benefits of the two proposals by using the messaging scenario of the DirectoryMark directory benchmark suite as input workloads. By using a system profiling technique, we confirmed that the performance bottlenecks were effectively eliminated by the proposed designs. The IDL cache and the IDL stack slab can improve the performance of the OpenLDAP directory server for a wide range of workloads having varying degree of searches to updates ratio.

As an on-going work, we also proposed the BER cache to speed up the BER encoding part of the OpenLDAP 2.1 directory server. Because its performance benefit is dependent heavily on the directory use patterns, we are planning to evaluate its performance with the traces obtained from a real enterprise-grade directory service.

In a memory constrained situation, relative sizes of the multiple caches in the system should be carefully determined for optimal performance. Moreover, the size of those caches needs to be adjusted on-the-fly, to cope with the changing workloads and access patterns, in a highly dynamic, on-demand computing environment. The effective allocation of memory among multiple caches is left as a further work.

We are also investigating performance improvements by the per-thread memory allocator and various connection management alternatives to improve performance further.

Acknowledgement

The authors thank the OpenLDAP community for invaluable comments and helps for performance improvement. Special thanks to Howard Chu of Symas Corp.

References

[1] ITU-T, "The directory: Overview of concepts, models, and service. recommendation X.500," International Telecommunication Union, 1993.
[2] The OpenLDAP Project, "OpenLDAP: Community developed LDAP software," http://www.openldap.org.
[3] ITU-T, "ASN.1 encoding rules: Specification of basic encoding rules (BER), canonical encoding rules (CER), and distinguished encoding rules (DER). recommendation X.690," International Telecommunication Union, Dec 1997.
[4] M. Wahl, T. Howes, and S. Kille, "RFC 2251 : Lightweight directory access protocol," Dec 1997.
[5] ITU-T, "The directory: Models. recommendation X.501," International Telecommunication Union, 1993.
[6] ITU-T, "Abstract syntax notation one (ASN.1): Specification of basic notation. recommendation X.680," International Telecommunication Union, Dec 1997.
[7] M. Wahl, A. Coulbeck, T. Howes, and S. Kille, "RFC 2252 : Lightweight directory access protocol (v3): Attribute syntax definition," Dec 1997.
[8] Gerald Carter, *LDAP System Administration*, O'Reilly, Mar 2003.
[9] Mindcraft, "DirectoryMark : The LDAP server benchmarking tool," http://www.mindcraft.com/directorymark.
[10] T. Howes, "An X.500 and LDAP database: Design and implementation," ftp://terminator.rs.itd.umich.edu/ldap/papers/ xldbm.ps.
[11] Dileep Bhandarkar and Jason Ding, "Performance characterization of the Pentium Pro processor," in *Proc. 3rd IEEE Int'l Symposium on High Performance Computer Architecture*. 1997, pp. 288–297, IEEE Press.
[12] Daniel P. Bovet and Marco Cesati, *Understanding the Linux Kernel*, O'Reilly & Associates, Inc., 2001.