Implementation and Evaluation of Moderate Parallelism in the BIND9 DNS Server

JINMEI, Tatuya
Toshiba Corporation
jinmei@isl.rdc.toshiba.co.jp

Paul Vixie
Internet Systems Consortium
vixie@isc.org

Abstract

Suboptimal performance of the ISC BIND9 DNS server with multiple threads is a well known problem. This paper explores practical approaches addressing this long-standing issue. First, intensive profiling identifies major bottlenecks occurring due to overheads for thread synchronization. These bottlenecks are then eliminated by giving separate work areas with a large memory pool to threads, introducing faster operations on reference counters, and implementing efficient reader-writer locks. Whereas some of the solutions developed depend on atomic operations specific to hardware architecture, which are less portable, the resulting implementation still supports the same platforms as before through abstract APIs. The improved implementation scales well with up to four processors whether it is operating as an authoritative-only DNS server, with or without dynamic updates, or as a caching DNS server. It also reduces the memory footprint for large DNS databases. Acceptance of this new sever will also have a positive side effect in that BIND9, and its new features such as DNSSEC, should get wider acceptance. The direct result has other ramifications: first, the better performance at the application level reveals a kernel bottleneck in FreeBSD; also, while the results described here are based on our experience with BIND9, the techniques should be applicable to other thread-based applications.

1 Introduction

As the Internet has become an indispensable piece of infrastructure, the domain name system (DNS)[10] has also been facing increasing pressure as a core feature of the Internet. For example, forged DNS data can lead to net scams, and the IETF has standardized a new version of security framework for DNS (DNSSEC[2]) in order to address such threats. At the same time, top level DNS servers have been receiving more queries, and have even been the target of denial of service attacks. Today's DNS servers thus need to have the latest functionality as well as higher performance to handle the heavy query rate.

ISC BIND9[6] was designed to meet these seemingly contradictory requirements. It supports all standardized DNS-related protocols, including the latest version of DNSSEC as well as adopting a multi-thread architecture so that it could meet the performance requirements by using multiple processors.

Unfortunately, BIND9's multi-thread support did not benefit much from multiple processors. The performance measured by queries per second that the server could process was soon saturated or even degraded as the number of processors and threads increased. In the worse case, BIND9 with multiple threads showed poorer performance compared to a single process of its predecessor, BIND8, which does not even try to take advantage of multiple processors.

Partly due to the poorer performance, operators who see high query rates have tended to stick to BIND8, implicitly hindering wider deployment of new technologies like DNSSEC.

We address the performance problem with multiple threads through a set of practical approaches and have already reported a preliminary result [7] for an authoritative-only DNS server that does not allow dynamic update requests [16]. This paper completes the ongoing work by improving memory management further and providing thorough evaluation and discussions of the implementation. The evaluation covers the case with dynamic updates or caching and with test data based on real DNS traffic.

The rest of the paper is organized as follows. In Section 2, we provide an overview of the BIND9 implementation architecture as a base for later discussions. Section 3 explains how to identify major bottlenecks regarding thread synchronization, and Section 4 describes our approaches to eliminate the bottlenecks followed by some detailed notes about the implementation in Section 5. We then show evaluation results for the improved implementation in terms of response performance and run-time memory footprint along with relevant discussions in Section 6. We review related work in Section 7 and conclude in Section 8 with remaining work.

2 BIND9 Architecture

In general, DNS servers are categorized as authoritative servers and caching (recursive) servers. An authoritative server has the authority for at least one *zone*, an administrative perimeter within the DNS. When the server receives a DNS query for a domain name, it searches the zone that best matches the queried name, and, if found, responds with the corresponding resource records (RRs) stored in the zone. A caching server handles queries for end stations by forwarding them toward authoritative servers until a determinate answer is received. This process is called *recursive resolution*. The answer is cached for possible re-use, and then forwarded back to the originating end station.

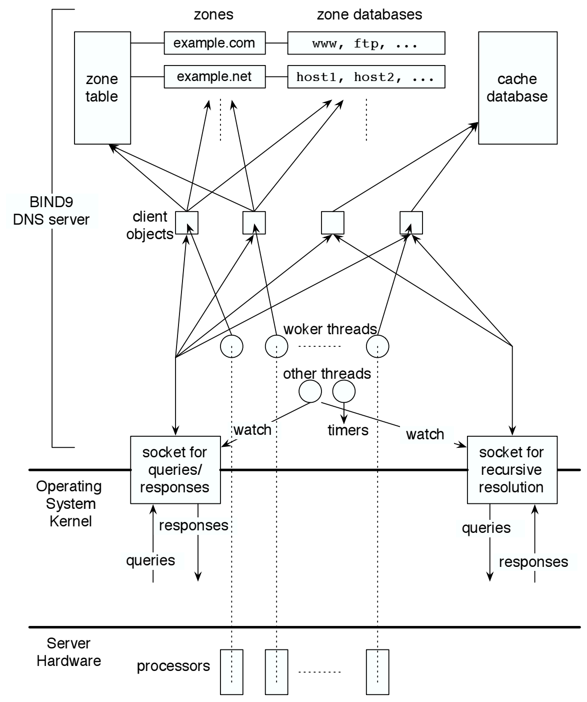

Figure 1: The system architecture of a DNS server with multiple processors running threaded version of BIND9.

Figure 1 is an overview of the entire system of a DNS server that has multiple processors and runs BIND9 with multiple threads.

BIND9 can act as either an authoritative or caching server. For the authoritative service, it has a *zone table*, whose entries correspond to the zones for which the server has the authority. Each zone entry has an associated *zone database*, which consists of RRs of that zone. For the caching service, it has a separate *cache database* for storing results of recursive resolution. BIND9 can support multiple implementations of databases, but the default implementation of zone databases and the cache database is the same. The zone table, zone databases, and the cache database are, by default, built in-memory, either from configuration files or dynamically.

When built with threads, BIND9 creates a set of *worker threads* at startup time. The number of worker threads is equal to the number of available processors by default, and the threads run concurrently on different processors handling DNS queries. Note that an additional "pool" of worker threads is unnecessary and indeed unused, especially for an authoritative server; since the processing of each query does not involve any network or file I/O, or blocking operations in general, there is no benefit in handling more queries than the number of processors concurrently.

Each DNS query arriving at the server is represented as a separate data structure called a *client object* in the BIND9 implementation. Available worker threads are assigned to the client objects and process the queries by consulting the zone table and the appropriate zone databases. If the server does not have authority for the queried domain name, the worker thread looks for the answer in the cache database. In either case when an answer is found the worker thread sends the response packet to the socket that received the query.

A caching server often needs to send additional queries to external authoritative servers in processing incoming queries. In BIND9 the worker thread sends such queries through the client object and receives the responses for further processing. While waiting for the responses, the worker thread works on other client objects, if any, so that it can handle as many queries as possible.

There are some other threads, regardless of the number of processors, which manage timers or watch events regarding sockets.

Since multiple threads get access to the zone table and databases concurrently, such access must be properly protected to avoid race conditions. BIND9 uses standard POSIX pthread locks [4] for this purpose.

3 Identifying Bottlenecks

The first step in this work was to identify major bottlenecks that affected BIND9's response performance in multi-processor environments. We used a simple profiler for lock overhead contained in the BIND9 package for this purpose. It is available by building the server with the `ISC_MUTEX_PROFILE` preprocessor variable being non-0. When enabled, it measures for each lock the waiting period for achieving the lock and the period that the lock is held by issuing the `gettimeofday` system call before and after the corresponding periods. The BIND9 server retains the statistics while it is running, and dumps the entire result on termination as follows:

```
mem.c   720: 1909760  7.223856 24.671104
 mem.c 1030:  108229  0.401779 1.625984
 mem.c 1241:      67  0.000107 0.000108
 ...
```

The above output fragment shows the statistics of a lock created at line 720 of `mem.c`. This lock was acquired 108,229 times at line 1030 of `mem.c`, the total time that threads spent in the corresponding critical section was 0.401779 seconds, and the total time that threads waited to acquire the lock was 1.625984 seconds.

We built BIND 9.3.2 on a SuSE Linux 9.2 machine with four AMD Opteron processors and with enabling the lock profiler [1]. We then configured it as a root DNS server using a published snapshot of the root zone, sent queries to the server for 30 seconds, and observed its behavior (see Section 6 for more details about the test environment).

According to the profiling results, threads spent about 52 seconds for acquiring locks, which were 43.3% of the total running time of the worker threads (120 seconds with four threads). We examined the results and categorized the points in the code that dominated the waiting periods as follows:

- 54.0% of the total waiting period belonged to memory management routines for allocating or releasing temporary memory to make response packets.

- 24.2% were mainly for incrementing or decrementing reference counters to some data objects.

- 11.4% were for getting access to data objects representing zones or resource records.

- 10.4% took place in BIND9's internal reader-writer lock implementation. These locks were for either the zone table or zone databases.

It may look too severe that nearly a half of the running time was occupied just for acquiring locks. But it is actually not surprising because the processing of DNS queries is lightweight. Especially in the case of an authoritative server, it does not involve any additional network or disk I/O, and the synchronization cost between the threads is relatively much expensive.

4 Eliminating Bottlenecks

In the following subsections, we discuss details about how to eliminate the bottlenecks shown in the previous section.

4.1 Working Space Separation

According to the profiling results, the most significant bottleneck was in memory management routines. Specifically, it was caused by a mutex lock for a data structure called the *memory context*. This structure maintains memory usage information in order to detect a memory leak and contains some small fragments of memory that can be reused. Multiple client objects share a single memory context and use it for building response messages frequently in the original BIND9 implementation, which caused the severe lock contentions.

In addition, the original implementation uses the standard library versions of `malloc` and `free` to allocate and release memory, which may internally cause additional lock overhead.

We mitigated the first type of overhead by separating memory contexts for different client objects. This is safe because the allocated memory is a temporary working space specific to each client and the worker thread working on it, and generally does not have to be shared with other threads.

In order to reduce the overhead imposed by the standard library, we enabled a BIND9's experimental feature called the "internal memory allocator". It initially allocates a 256KB memory block as a large pool for each memory context, and repeatedly uses fragments in the memory pool without involving the standard library.

One obvious issue with this approach is the additional memory consumption for the separate space. Overconsumption of memory is of concern particularly when BIND9 acts as a caching server. In this case, it can have many client objects that are handled by a smaller number of worker threads: by default, the maximum number of concurrent clients is 1000, while the number of worker threads is equal to the number of processors. If we simply separated the work space for each client object, the maximum amount of memory needed might be unacceptably large.

Instead, we had multiple client objects share a single memory context as shown in Figure 2. The number of contexts has a fixed limit regardless of the number of client objects, thereby ensuring the upper limit of required memory for the work space.

Since different worker threads can work on client objects sharing the same memory context, access to the allocated memory must be locked properly. For example, worker threads A and B in Figure 2 share the same mem-

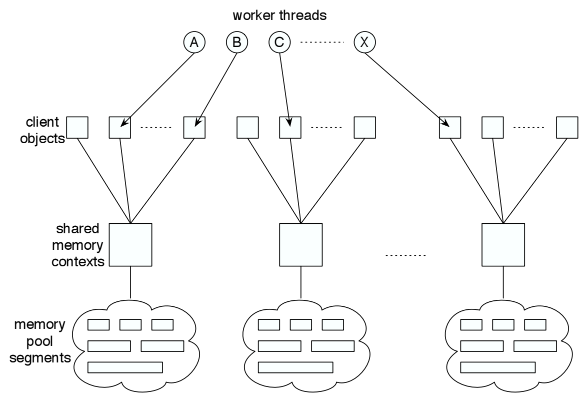

Figure 2: Sharing memory contexts by clients.

ory context and may contend with each other for holding the lock of the context.

We avoided the lock contention in a probabilistic way, that is, by creating many more shared memory contexts on-demand than worker threads. In the current implementation, the maximum number of memory contexts is 100, which we believe to be large enough for a decent number of worker threads in environments where BIND9 is expected to work. Yet this number is an arbitrary choice; in fact, we have not seen significant differences in performance with 10 or 100 clients with evaluation configurations described in Section 6. Its effectiveness will have to be assessed through live experiments.

Regarding the additional memory consumption, the total amount of memory for the preallocated block occupies the dominant part. If all the 100 contexts are created and used, the total amount will be 25MB. More blocks of memory could be required if the clients consume all the pre-allocated memory, but typically it should not happen since the memory is used only for temporary work space to build a moderate size of response message and only requires a small amount of memory. It should also be noted that an authoritative server does not need additional clients, and the discussion only applies to a caching server. For a busy caching server, which generally needs a large amount of memory for the cache, we believe the additional consumption is minor.

4.2 Faster Operations on Counters

As shown in the previous section, nearly a quarter of the major bottlenecks in terms of the waiting period caused by locks were regarding operations on reference counters. Reference counters are essential for threaded programs, since multiple threads can independently refer to a single data object and it is hard to determine without counters whether a shared object can be safely destroyed.

Obviously, change operations on a reference counter must be mutually exclusive. In addition, a significant action for the referenced object can happen when the reference increases from or decreases to zero, in which case additional locks may be necessary to protect that action. However, such an event does not take place in the typical case where worker threads are working on those references, since the base system usually holds the structure in a stable state. For example, a zone keeps a positive reference to the corresponding database until it replaces the database with a new one, e.g., by reloading the configuration file or at shutdown time. Thus, the locks for reference counters generally only protect the simple operation of increasing or decreasing an integer.

This observation led us to the following idea: most of today's processors have dedicated atomic instructions that allow multiple threads to increment or decrement an integer concurrently and atomically. Some processors support an instruction exactly for this purpose. We can also implement these operations with other type of instructions such as compare-and-swap (CAS) and a busy loop, as shown in Chapter 15 of [8]. In either case, the resulting operations on reference counters can run much faster than those protected by general locks, and we will simply call both "atomic operations" hereafter. We still use normal POSIX locks for the rare cases where the counter decrements to zero.

4.3 Efficient Reader-writer Lock

According to the results of Section 3, about 30% of the locks affecting the performance were related to access to the zone table or to zone databases. While some of the locks are general and allow only one thread to enter the critical section at once, the access is read-only in most cases for an authoritative DNS server that does not allow dynamic updates: once the server loads the configuration and zones, the worker threads just search the zone table and zone databases for the answers to queries, without modifying the objects they are referring to.

One possible optimization here is to use reader-writer locks (rwlocks). In fact, the original BIND9 implementation used rwlocks for some cases considered here. It uses a custom version of rwlocks instead of the pthread rwlock library in order to ensure fairness, and the custom implementation relies on the general locks internally. As a side effect of this, readers can be blocked even if there is no writer.

If we could assume some atomic operations supported by the underlying hardware, however, rwlocks could be implemented more efficiently [9]. We adopted a slightly modified version of the "simple" algorithm described in [9] [2], and used the new rwlocks for all of the above cases. The new rwlock implementation requires an atomic operation that adds some (positive or negative) value to an integer and an atomic CAS operation.

The key to the algorithm are the following integer variables, which are modified atomically: write_requests, write_completions, and cnt_and_flag.

The first two variables act as a waiting queue for writers in order to ensure the FIFO order. Both of them begin with the initial value of zero. When a new writer tries to get a write lock, it increments write_requests and gets the previous value of the variable as a ”ticket”. When write_completions reaches the ticket number, the new writer can start writing. When the writer completes its work, it increments write_completions so that another new writer can start working. If the write_requests is not equal to write_completions, it means a writer is now working or waiting. In this case, new readers cannot start reading; in other words, this algorithm prefers writers.

The variable cnt_and_flag is a “lock” shared by all readers and writers. This 32-bit integer variable acts as a structure with two members: writer_flag (1 bit) and reader_count (31 bits). The writer_flag bit reflects whether a writer is working, and reader_count is the number of readers currently working or almost ready to work (i.e., waiting for a currently working writer).

A writer who has the current “ticket” tries to get the lock by exclusively setting the writer_flag to 1, provided that the whole 32-bit value is 0 (meaning no readers or writers working). We need the atomic CAS instruction here. On the other hand, a new reader first checks there are no writers waiting, and then increments the reader_count field while getting the previous value of cnt_and_flag. If the writer_flag bit is not set, then the reader can enter the critical section; otherwise, it waits for the currently working writer.

When the necessary prerequisite conditions are not met, the reader or the writer sleeps until the related conditions change. When a working reader or writer completes its work, some readers or writers are sleeping, and the condition that suspended the reader or writer has changed, then it wakes up the sleeping readers or writers. Our implementation uses condition variables and locks as defined in the standard pthread library for these cases. The use of the standard library may degrade the performance, but we believe this does not matter much, since writers should appear only very rarely in our intended scenarios. In addition, we found the extensions to the original algorithm described below could be implemented much easier with the standard library for the rare cases.

The original algorithm based on [9] was not starvation-free for readers. In order to prevent readers from starving, our implementation also introduced the “writer quota” (Q). When Q consecutive writers have completed their work, possibly suspending readers, the last writer will wake up the readers even if a new writer is waiting.

We implemented other extensions: “trylock”, “tryupgrade”, and “downgrade”, which are necessary in some special cases in the BIND9 implementation.

The “trylock” extension allows the caller to check whether the specified lock is achieved and to get the lock when possible without blocking. This extension is actually used for writers only. Our implementation of this extension lets the caller pretend to have the “current ticket” and try the same thing as a candidate writer would normally do as described above. If this succeeds, then the caller decrements write_completions as if it had the “current ticket”, and starts writing. Otherwise, this attempt simply fails without making the caller sleep.

The “tryupgrade” extension allows a reader who already has a lock to become a writer if there are no other readers. The implementation is similar to that of “trylock”, but in this case the prerequisite condition for the CAS operation is that reader_count be one, not zero, which indicates this is the only reader.

On the other hand, if a writer “downgrades”, it becomes a reader without releasing the lock, so that other readers can start reading even if there is a waiting writer. The implementation of downgrading is straightforward and is not discussed here.

4.3.1 Alternative Approach: Standard Rwlock

One possible alternative to the optimized, but less portable rwlock described above is the standard rwlock implementation [15] provided as a part of the operating system. It is clearly advantageous in terms of portability, and it may also provide decent performance depending on the implementation detail.

However, the standard rwlock specification is not guaranteed to be starvation free, which is the primary reason why we did not rely on it. In addition, the performance of the standard implementation varies among different implementations as we will see in Section 6.1.6. Considering the variety of architectures that have the necessary atomic operations for the customized rwlock implementation (see Section 5.2), we believed our approach with the customized implementation would provide better performance for as many systems as possible.

5 Implementation

We modified BIND9 using the optimization techniques described in the previous section and contributed the new implementation to ISC. While it is not publicly available as of this writing, it has been incorporated in the ISC's development tree, and will be released as free software in BIND 9.4, the next major release of BIND9.

In the following subsections we make some supplemental notes on the implementation.

5.1 Where and How to Use Rwlocks

It was not straightforward to apply the efficient rwlock implementation described in Section 4.3. We used the rwlocks for the zone table, each of the zone and cache databases, and every database entry, which is sets of RRs (called *RRsets*) for the same name.

The original implementation used normal mutex locks for the RRsets, not rwlocks. In order to use the rwlocks for the RRsets in an effective way, we first separated the targets protected by the locks into reference counters and other content. We then ensured safe and efficient operations on the former as described in Section 4.2 while using rwlocks for protecting access to the latter. This way we typically only need read-only access to the RRsets, where the rwlocks work effectively. This is definitely the case for authoritative-only servers that do not allow dynamic updates.

For caching servers, which need write access to RRsets in the cache database more frequently, we introduced one further customization. Whether it is a zone or cache database, the original implementation used a lock bucket, each entry of which contained a lock shared by multiple RRsets in order to limit the necessary memory for the locks. Moreover, the number of entries in a bucket was set to 7 by default, a pretty small number, so that the memory consumption would still be small even if the server had a large number of zones. Using a lock bucket should make sense, but it was not reasonable for applying the same limit of the bucket size to zone databases and the cache database because there is typically only one cache database in a single server. Thus, we enlarged the bucket size for the cache database to a much larger number. It is 1009 in the current implementation, which is an experimental arbitrary choice at the moment.

5.2 Portability Considerations

Some approaches described so far rely on hardware dependent atomic operations. An obvious drawback of such approaches is that the resulting implementation becomes less portable. This is particularly severe for BIND9, since portability is one of its major goals as a reference implementation.

To ensure as much portability as possible, we minimized the necessary operations. In fact, we only need two operations: atomic addition on an integer, which can also be used for atomic operations on counters, and atomic CAS. Furthermore, these can actually be emulated using other primitive operations.

We have implemented the two operations, through emulation in some cases, for Intel x86, AMD, Alpha, Sparc v9, PowerPC, and MIPS as a proof of concept. The first two have dedicated instructions for our purposes [5]. Sparc v9 has CAS as a primitive instruction [13], on which we can emulate the atomic addition on an integer. The others have a variant of locked load and store instructions, with which we can emulate the necessary operations. We believe these operations can also be implemented on many of today's architectures.

However, we still expect the operations cannot be provided in a realistic way for some architectures. Thus, any use of these operations are hidden under an abstract interface, and the new code works just as before on machines that do not support the necessary operations for our optimizations.

5.3 Optimizing OS Kernel

Application-level optimization sometimes reveals system bottlenecks. In fact, after implementing the possible optimizations described so far, we noticed that FreeBSD still did not show the performance we expected on a fast multi-processor machine, while Linux and Solaris showed the anticipated improvement.

We then examined the kernel source code of FreeBSD and found that a kernel lock was protecting the send socket buffer. The lock was to protect simultaneous access to parameters of the socket buffer such as watermarks for flow control or a list of outgoing packets kept in the socket for retransmission. This lock was held throughout the socket output routine including the output processing at the network and datalink layers. Since a DNS server typically works on a single UDP port and the worker threads of BIND9 share a single UDP socket for sending responses as a result, they would contend in the kernel attempting to acquire the lock.

For a UDP socket, however, this lock was unnecessary. In fact, none of the protected parameters were used for the output processing of UDP packets due to its property as a non-reliable transport protocol without flow control.

We wrote a simple patch that omitted the unnecessary lock for UDP sockets, and confirmed that it worked efficiently without causing any disruption in the kernel. We then reported this performance issue with the patch to the FreeBSD developers community. Fortunately, FreeBSD's netperf project [17] has removed this bottleneck in a recent change based on our report.

6 Evaluation

We evaluated the effectiveness of the implementation described so far using a real multi-processor machine and actual DNS queries.

The base of our implementation was an unpublished version of BIND9. The response performance we achieved, however, should be equivalent to BIND 9.3.2, the latest released version as of this writing.

The target machine used for the evaluation was a 4-way AMD Opteron with 3.5GB of memory, 1GB of L2 cache, and 64KB of L1 cache (64KB each for instruction and data) and had a Broadcom BCM5704C Dual Gigabit Ethernet adaptor.

Most of the results shown in this paper were those for SuSE Linux 9.2 (kernel 2.6.8 and glibc 2.3.3) running in the 64-bit mode, unless explicitly noted otherwise. We also performed the same tests against FreeBSD 5.4-RELEASE and Solaris 10 on the same hardware, both running in 32-bit mode, and confirmed the overall results were generally the same.

We built BIND9 both with and without the optimizations we developed on the test machine, which we call "BIND9 (old)" and "BIND9 (new)", respectively. For comparison, we also used BIND9 without enabling threads and BIND 8.3.7. These are referenced as "BIND9 (nothread)" and "BIND8".

6.1 Response Performance

6.1.1 Server Configurations

We configured various types of DNS servers for evaluating the response performance of our implementation. These configurations have several different characteristics.

The first type of configuration is for "static" authoritative servers, i.e., authoritative-only DNS servers that do not allow dynamic updates, which has the following, three specific configurations:

- A root server configuration with a real snapshot of the root zone as of October 28th, 2005. Snapshots of the root zone are publicly available at ftp://ftp.internic.net/. We specifically emulated the F-root server, which also had authority for the ".ARPA" zone. We used a copy of the ".ARPA" zone data of the same day from the F-root server via the zone transfer protocol and used it for configuring the test server.
- The ".NET" server configuration with a copy of the actual zone database as of March 2003. It contained 8,541,503 RRs and was used as a sample of a single huge zone. The vast majority of the zone data consists of NS and glue RRs.
- A generic server configuration with a massive number of small zones. Specifically, it had 10,000 zones, each of which contained 100 A RRs in addition to a few mandatory RRs for managing the zone.

The second type of configuration is for a "dynamic" authoritative server. It started with a single pre-configured zone containing 10,000 RRs in addition to a few mandatory RRs and accepted dynamic update requests.

The third type of configuration is for a "caching" recursive server. It started without any authoritative zones and accepted recursive queries.

6.1.2 Evaluation Environment

Our measure of performance common to all the configurations was the maximum queries per second that the tested implementation could handle without dropping any queries. Our engineering goal regarding this measure was to add 50% of the single-processor query rate for every additional processor. This engineering goal was set so that BIND9 with two processors would outperform BIND8 if BIND9 with a single processor could operate at no less than 80% of the speed of BIND8 on that same single processor. While the target may sound conservative, we believe this is in fact reasonable since the base performance of BIND9 is generally poorer than BIND8 due to its richer functionality, such as support for DNSSEC, and the cost of securer behavior, such as normalization of incoming data.

We connected two external machines to the same Gigabit Ethernet link to which the test machine was attached. The external machines acted as the clients for the evaluation target. In order to avoid the situation where performance on the client side was the bottleneck, we used multiple processes on the client machines.

We used a variant of the `queryperf` program contained in the BIND9 distributions for the client side processes, which was slightly customized based on the original tool for our testing purposes. The program was modified so that it could combine the results of multiple processes on different machines and could send dynamic update requests as well as ordinary queries. In all test cases `queryperf` repeatedly sent pre-configured queries, adjusting the query rate in order to avoid packet loss, and dumped the average queries per second processed.

6.1.3 Static Server Performance

We prepared the test queries for the root server configuration based on real traffic to the F-root server located in San Francisco, California. We used packets that had reached the server between 5:18am and 6:18am on Oc-

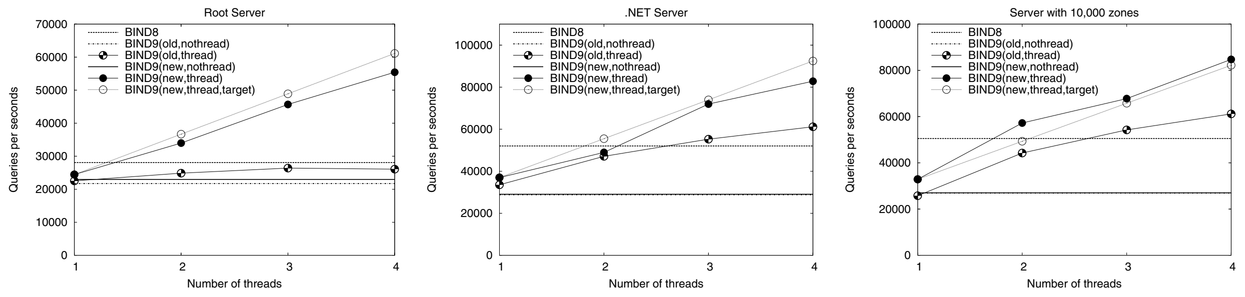

Figure 3: Evaluation Results for the static configurations.

tober 28th, 2005, a sample which contained 31,244,014 queries.

The reason why we performed the evaluation based on real data was because details of the query pattern may affect the response performance significantly. Of particular importance in terms of bottlenecks for a multi-threaded BIND9 server is the proportion of queries that match the same database entries; if a large fraction of the whole queries matches a small number of entries, it may cause contentious access by multiple threads and can be a bottleneck due to the contentions for acquiring locks. In fact, we analyzed the query data and found that the query pattern was unbalanced: 22.9% of the queries was for names under the same domain (.BR) and names under the top six domains occupied more than a half of the whole queries.

We separated the queries into three chunks and ran three instances of the `queryperf` program concurrently with the divided queries as input so that the resulting test query stream roughly emulated the actual query traffic. We did not stick to reproducing the real queries in the same order and at the same timing since our primary goal was to know the maximum performance of the tested implementation.

For all other authoritative server configurations than the root server case, we generated the test queries by randomly choosing domain names under the zone that the target server managed. A small fraction of the query names did not exist in the zone and resulted in negative responses.

Figure 3 summarizes the evaluation results. For BIND8, and for BIND9 with “nothread”, the results for more than one thread are meaningless. For comparison purposes, however, we showed these meaningless results as if they were independent of the number of threads. The graphs labeled “target” show our original engineering goal.

Although we did not reach our original engineering goals in some cases, our results are generally good. The new BIND9 implementation also scaled almost linearly to the number of processors, up to a maximum of four processors.

In comparison to BIND8, the new BIND9 implementation with two processors could answer the same number or more queries than BIND8 could, and with three or more processors the results were much better than BIND8. We also note that the old BIND9 implementation could never outperform BIND8 even with all the four processors for the root server configuration.

Another remarkable point is that BIND9 with one thread could generally handle more queries than BIND9 without threads. This is not an intuitive result, since both implementations should benefit only from a single processor and there should be overhead specifically related to threads. We discovered the reason for the result was because the “nothread” version needed to check other asynchronous events such as timer expiration periodically, even though such events rarely occurred; in the threaded version, the worker thread could keep handling queries without any interrupts, since separate threads which were effectively idle could handle the exceptional events. This should prove that a well-tuned threaded application can run more efficiently even on a single processor machine.

6.1.4 Performance with Dynamic Updates

To evaluate the authoritative server while it was allowing dynamic updates, a separate client process sent update requests at a configured rate, cooperating with the `queryperf` processes. In order to emulate a busy authoritative server handling frequent update requests, we measured the total response performance using the update rates of 10 and 100 updates per second.

Figure 4 is the evaluation results of these scenarios. There is an exception in the case of BIND9 implementations receiving the higher rate of update requests with no or one thread: the BIND9 server using a single processor could only handle at most 75 updates per second. With two or more processors, it could process the configured update rate.

In either case the new BIND9 implementation

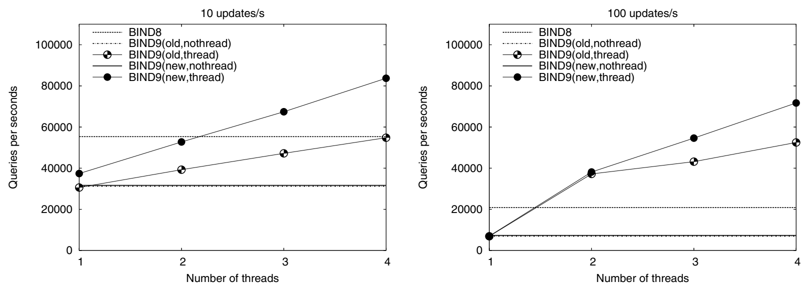

Figure 4: Evaluation results for a server accepting 10 and 100 update requests per second.

achieved the expected improvement in that it could at least handle with two threads as many queries as BIND8 and could process more with three or more threads. This result proves that the efficient rwlock implementation works as expected with a small number of writers.

It should be noted that dynamic update processing cannot benefit from multiple processors, since update requests must be handled sequentially. We thus fixed the update rate regardless of the number of threads in the evaluation, which means the total queries handled with one thread is much smaller than the other cases. Therefore, the “target” performance in these cases do not matter much, and we did not include it in Figure 4.

6.1.5 Caching Server Performance

It is not trivial to evaluate the response performance of a caching DNS server as pointed out by [12]. The main reason is that query processing at a caching server can involve additional transactions with external authoritative servers and so the total performance depends on various factors such as available network bandwidth, the response performance or availability of the external authoritative servers, or the cache hit rates. Some of the dominant factors are hard to control or reproduce.

We used a simplified model for our evaluation as shown in Figure 5. It consisted of two external authoritative servers in addition to a client (tester) and the caching server (evaluation target), all attached to the same Ethernet segment. External server 1 had authority for 1,000 external zones, each of which contained 200 RRs, and external server 2 had authority for a common parent zone of these zones and delegated the authority of the child zones to server 1. The client sent queries for names belonging to these zones to the caching server.

In the initial stage of the performance evaluation the caching server needed to follow the delegation chain of authority from the root server to external server 1. The caching server then repeatedly queried external server 1 and cached the result as it received queries from the client (as shown by exchanges 1 through 4 in the figure). At

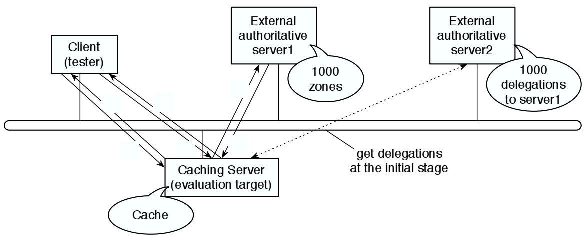
Client (tester)
External authoritative server1
1000 zones
External authoritative server2
1000 delegations to server1
get delegations at the initial stage
Caching Server (evaluation target)
Cache

Figure 5: Network configuration for evaluating caching server performance.

some point the cache stabilized, and the caching server did not have to query the external server as long as the data remained in the cache (as shown by exchanges 5 and 6 in the figure). At this stage we measured the response performance.

The configurable evaluation parameter in this test scenario was the cache hit rate. We configured half of the RRs stored in external server 1 with a Time-to-live (TTL) of 1 second so that it would be highly unlikely to be reused in the cache. If the caching server received a query whose answer had expired during the test period, the caching server queried the external server again. We then prepared various test query patterns for specific hit rates by mixing the names with the shorter TTL and the other names in an appropriate ratio.

Our primary target of cache hit rate was 80% hits. This number was based on a statistic analysis of an existing busy caching server that had a large number of clients. We also used other query data that caused the hit rates from 50% to 90% for comparison.

Figure 6 shows the evaluation result for the caching server configuration with the primary target of cache hit rate. While the new BIND9 implementation scaled well compared to the target performance, and it could answer more queries with four threads than BIND8, this was not fully satisfactory in that it needed all four threads (processors) to outperform BIND8.

It should be noted, however, that BIND9 with one thread could only handle 43% of queries that BIND8 could answer. This means that the fundamental reason for the poorer performance is the base processing ability with a single thread, as explained in Section 6.1.2, rather than bottlenecks in the multi-thread support. It should be more reasonable to consider improving the base performance first in order to achieve competitive performance with multiple threads in this case.

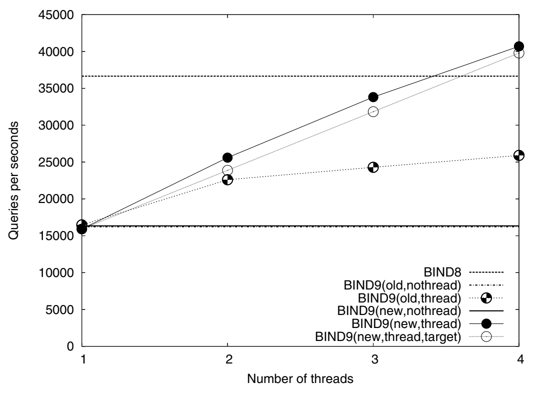

Figure 6: Evaluation Results for a caching server with 80% cache hits.

Table 1 summarizes the results of the same evaluation with various cache hit rates. It generally indicates the same result as shown in Figure 6 regardless of the hit rates: the new BIND9 implementation scaled well with multiple threads, but the resulting performance compared to BIND8 was not enough due to poorer base performance.

Rate (%)	BIND8	BIND9 (new) with threads 1	2	3	4
90	41393	20569 (50%)	31287 (76%)	**42826** (103%)	**51686** (125%)
80	36647	15915 (43%)	25595 (70%)	33809 (92%)	**40697** (111%)
70	32660	13388 (41%)	21638 (66%)	27561 (84%)	**33269** (102%)
60	29862	10296 (35%)	18059 (60%)	22834 (76%)	28100 (94%)
50	29676	9641 (32%)	15610 (53%)	19713 (66%)	24391 (82%)

Table 1: A summary of performance evaluation on caching server implementations with various cache hit rates (the left-most column). The numbers for the BIND8 and BIND9 columns indicate the maximum number of queries that the corresponding implementation could handle per second. The percentage numbers for the BIND9 columns are the ratio to the corresponding result of BIND8. The results of BIND9 better than BIND8 are highlighted in a bold font.

6.1.6 Comparison with Alternatives

We also performed additional tests with some of the above configurations for comparing several implementation options. The primary goal in these tests was to see whether the standard rwlocks can be used as a practical alternative as discussed in Section 4.3.1.

Figure 7 shows a summary of the performance comparison. All the optimization types utilize the portable technique of separate memory contexts described in Section 4.1. Note that optimization types (3) and (4) do not rely on hardware dependent atomic operations and are more portable than others. It should also be noted that database entries are protected by mutex locks, not rwlocks, in this type of optimization because the use of rwlocks for the database entries needs faster operations on reference counters as we explained in Section 5.1.

Figure 7 provides several lessons regarding the tradeoffs between the optimization techniques.

First, the difference between cases (A) and (B) shows the efficiency of the standard rwlocks varies among different implementations when only reader locks are necessary in the typical case. This means a portable application running on various systems that requires higher performance cannot always rely on the standard rwlock implementation.

Secondly, the result of case (A) indicates that simply using efficient rwlocks may not be enough in terms of total performance. In fact, the difference between optimization types (1) and (2) shows the standard rwlock library is as efficient as our customized version. The comparison between these two and optimization type (3), however, proves that we still need the help of atomic operations to achieve the required performance. One possible explanation of the difference is that type (3) had to use normal mutex locks shared in a small bucket for protecting access to database entries, causing heavy contentions. However, the comparison between types (3) and (4) rather proves that this is likely due to the unbalanced query pattern to the server as explained in Section 6.1.3.

Since the major benefit of implementation (3) is better portability by avoiding machine dependent operations while realizing better performance, this result indicates that attempting to use standard rwlocks cannot completely achieve that goal.

Finally, test cases (C) and (D) seem to show it is not very advantageous to use efficient rwlocks, whether they are based on atomic operations or the standard implementation. In fact, even optimization type (4) worked nearly as efficient as type (1). This is likely because efficient rwlocks were less effective in this case due to the more frequent write operations and because access to cache database entries was less contentious thanks to the

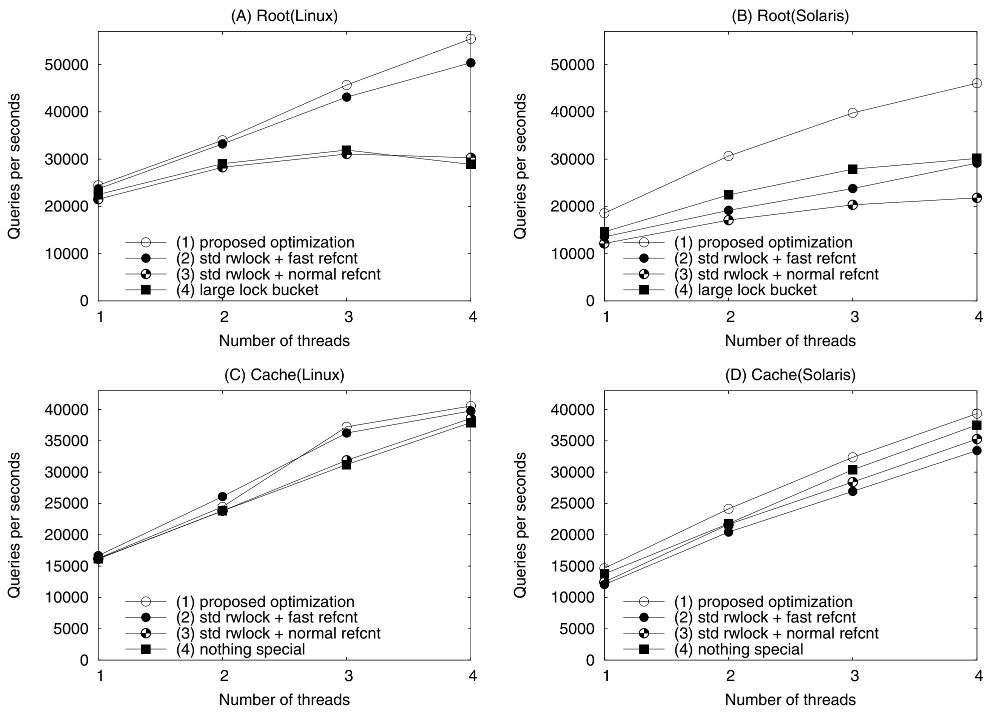

Figure 7: Performance comparison between various types of optimization for the root and caching servers on Linux and Solaris. Optimization types (1) to (3) are common to all the test cases: (1) is the proposed version in this paper; (2) utilizes the standard rwlocks (Section 4.3.1) and faster operations on reference counters; (3) uses the standard rwlocks and normal reference counters. For the root server configuration, optimization type (4) uses a larger lock bucket for zone databases but does not rely on other optimizations, while type (4) for the caching server configuration does not benefit from any optimization.

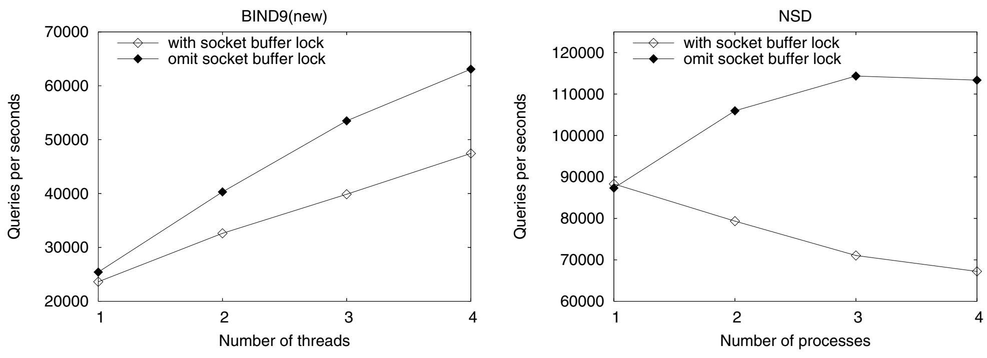

Figure 8: Evaluation of the FreeBSD kernel optimization for the root server configuration.

larger lock bucket.

It should be noted, however, that the actual query pattern may change the result. We generated the query input by randomly and equally choosing names from a large set in these tests, which means query names were well-distributed and less likely to cause contention when acquiring locks. If the real query pattern is more unbalanced as we saw in the root server case, it may cause severe contention that can make a more visible difference in the response performance depending on the rwlock implementation. In fact, we believe this to be the case even for a caching server. For example, the host names of popular search engines stored in the cache database may be queried much more frequently than others. We need further tests with input data which better emulates the real traffic in order to get a definitive conclusion.

6.1.7 Effect of Kernel Optimization

Finally, we verified whether the kernel optimization for FreeBSD described in Section 5.3 made a notable difference with actual query traffic. In addition to the optimized version of BIND9, we used NLnet Labs' NSD[11] for this test. Unlike BIND9, NSD forks multiple processes for concurrent operation, all of which share a single UDP socket. By using NSD, we were able to eliminate the possibility of thread-related bottlenecks altogether.

Figure 8 shows the evaluation results with the root server configuration with or without the lock for the UDP socket output buffer. We omitted the results of the other authoritative server scenarios, but they generally indicated the same conclusion. We also did not show the caching server case because the possible performance was not so high to reveal the kernel bottleneck.

The evaluation results clearly show that the unnecessary lock was a severe bottleneck for a high-performance application that shares a single UDP socket among multiple threads or processes. In particular, the result for NSD indicates the bottleneck can even degrade the performance with a larger number of processes.

While the performance of NSD was saturated with three or more processors, it was not specific to FreeBSD; we saw similar results on Linux and Solaris. We did not figure out the reason for this, but speculated this was due to some lower-level bottlenecks such as in the physical network interface or in memory access contentions.

We did not directly compare the performance between BIND9 and NSD in this test. In fact, the comparison would not make much sense since NSD concentrates on achieving higher response performance by omitting some richer functionality that BIND9 can provide. We will give a qualitative comparison between these two implementations in Section 7.

6.2 Memory Footprint

There is no protocol dictated upper limit of the size of a DNS zone. In fact, some top level zones contain more than 10 million RRs. As a result, the size of DNS zone databases for a server implementation can be huge. For implementations that store the databases in-memory, including BIND, run-time memory footprint is thus critical.

Our implementation adds a new structure field to a set of RRs for faster operations on reference counters, and could increase the memory footprint proportional to the number of the sets, which is typically proportional to the number of RRs. Additionally, we use the efficient version of rwlocks for more data objects than the original implementation did. Since a rwlock generally requires more memory than the normal lock, this could also be a source of larger memory footprint.

We therefore assessed the needed memory for some typical cases which require more memory: a case of a huge zone and a case of 10,000 zones, which were actually the second and third "static" configurations described in Section 6.1.1.

Table 2 summarizes the results. Against our expectation, the new implementation showed even better results than the old one in terms of memory footprint. We thus enabled BIND9's internal memory allocator (see Section 4.1) for the old implementation as well, and compared the results, which are also included in Table 2. It likely indicates the internal allocator recycles memory fragments effectively and reduces the total memory consumption. This also means that the use of internal allocator in our approach makes up for the additional memory consumption used in reducing the lock overhead.

	FreeBSD(32bit)		Linux(64bit)	
Config	Old	New	Old	New
".net"	762(562)	583	907(802)	811
10K zones	174(143)	164	230(200)	221

Table 2: Run-time memory footprint of authoritative servers of the old and new BIND9 implementations (in MB). Numbers surrounded by parentheses are the footprints for the old implementation enabling internal memory allocator.

7 Related Work

A major subject of our work is to make synchronization among multiple threads more efficient. This is a well-understood research topic, and, indeed, the efficient reader-writer lock implementation we described in Section 4.3 was based on a simple and "naive" algorithm mentioned in old work[9].

As summarized in [1], the current trend of this research area is to pursue scalability with a much larger number of processors, especially using the notion of local spinning. From a practical point of view, however, the scalability aimed for in the theoretical field is far more than enough. In fact, even 4-way machines are not common in such practical areas as DNS operation. We showed in the previous section that our approach scaled well up to the reasonable upper limit.

Meanwhile, such scalable solutions tend to need more complex data structures, requiring more memory, and in some cases more atomic operations. Since some of the data objects we want to protect are sensitive to memory consumption, complex data structures are not really appropriate even if those provide highly scalable performance. Also, requiring more primitive operations damages portability, which is not acceptable from a practical point of view.

Whereas we adopted the built-in memory allocator of the BIND9 implementation for managing temporary work space as explained in Section 4.1, there are other scalable allocators intended for multithreaded applications such as Hoard[3]. It utilizes per-thread heaps to which a set of memory blocks are assigned as a dedicated memory pool for the corresponding threads. The use of per-thread heaps helps avoid "false sharing" (where multiple threads share data on the same CPU cache), yet it limits memory consumption regardless of the number of processors by recycling unused blocks depending on the total usage. In theory, Hoard is better than the BIND9 allocator since the latter can cause false sharing when multiple threads share a single memory context.

However, when we repeated the evaluation tests with BIND9, linking Hoard as well as enabling other optimizations[3], we found that it actually performed slightly worse than BIND9's internal allocator: the former ran 13.1% slower than the latter in the scenario of a caching server with 50% cache hits, which should be the severest test case for a memory allocator. This was probably because the scalability advantage of Hoard did not outweigh its internal complexity; we will have to evaluate the performance with a larger number of processors for fair comparison.

Regarding server implementations, the Apache HTTP server[14] uses atomic operations for incrementing and decrementing reference counters. To the best of our knowledge, however, specific performance evaluation has not been publicly provided. It is also not clear whether the introduction of the atomic operations was based on performance analysis. On the other hand, we first identified operations on reference counters were actually a severe bottleneck through profiling, and showed it could be resolved by introducing an atomic operation.

NSD[11] is another example of high performance DNS server. It makes the query processing fast by precomputing the image of response packets for typical queries at initialization time, assuming "static" zone configurations. NSD can also benefit from multiple processors by forking multiple processes for concurrent operation. Indeed, it scales well with multiple processors as we saw in Section 6.1.7. However, it has its own drawbacks. Since each process has an independent copy of data in memory, NSD cannot allow a part of a zone to be dynamically modified via the DNS dynamic update protocol or act as a caching server. Additionally, this approach is not suitable for managing a huge zone because the total memory footprint needed is proportional to the number of processors. Overall, the difference between BIND9 and NSD is a design tradeoff between richer functionality such as the support for dynamic update and higher possible performance in some limited environments.

8 Conclusions and Future Work

We explored several practical approaches for improving the responsiveness of the ISC BIND9 DNS server with multiple threads. We first identified major bottlenecks occurring due to overhead related to thread synchronization through intensive profiling. We then eliminated all the bottlenecks by giving separate work areas to threads using the notion of shared memory contexts, introducing faster operations on reference counters, and implementing efficient reader-writer locks (rwlocks). While some of the solutions developed depended on atomic operations specific to hardware architecture, which are less portable, the resulting implementation still supported the same platforms as before through abstract APIs. We confirmed our implementation scaled well with up to four AMD processors under various configurations from authoritative-only to caching, and with or without allowing dynamic update requests.

Our primary contribution is performance improvements in BIND9, a long-standing issue with that version which has prevented wider deployment. We also hope, as a consequence of these results, that this will promote deployment of new protocol features which previous major versions of BIND do not support, such as DNSSEC.

While the approaches we adopted specifically targeted one particular implementation, we believe our approach includes several lessons that can also help develop other thread-based applications. First, even the seemingly naive approach for identifying bottlenecks in fact revealed major issues to be fixed as shown in Section 3. Secondly, the fact that operations on reference counters were a major bottleneck is probably not specific to BIND9, since these are inherently necessary for objects shared by multiple threads. Thus, our approach to improve the performance for this simple and particular case will help other applications. Finally, the efficient implementation of rwlocks and the framework of shared memory pools can easily be provided as a separate library, not just a subroutine specific to BIND9, and help improve performance of other applications such as HTTP servers or other general database systems than DNS.

We also identified a bottleneck in the UDP output processing of the FreeBSD kernel through our attempt of improving and evaluating the target application and provided a possible fix to the problem. Other applications that benefit from the techniques we provided in this paper may also be able to reveal other bottlenecks hidden inside the system so far.

Even though we proved the effectiveness of our approach through pragmatic evaluation, there may be issues in the implementation which can only be identified with further experiments in the field. In particular, we need feedback on other machine architectures than that we used in Section 6, especially about scalability with a large number of processors, to assess the cost of emulation mentioned in Section 5.2. A larger number of processors may also reveal performance issues in the memory allocator, and will give a more reasonable comparison with a scalable allocator such as Hoard.

Some tuning parameters we introduced are currently an arbitrary choice (Sections 4.1 and 5.1), and we will need to evaluate their effectiveness. We also need more realistic tests for the caching server configuration in order to determine the most reasonable optimization technique regarding both performance and portability, as we discussed in Section 6.1.6. We will continue working on those areas, identifying and solving issues specific to such cases.

Acknowledgments

This work was supported by the Strategic Information and Communications R&D Promotion Programme (SCOPE) of the Ministry of Internal Affairs and Communications, Japan. It was also partially supported by NSF research grant #SCI-0427144.

We thank our shepherd, Vivek Pai, and the anonymous reviewers for their valuable comments and suggestions. We are also indebted to ISC engineers, especially to Mark Andrews, Rob Austein, João Damas, Michael Graff, David Hankins, Peter Losher, and Evan Zalys-Geller for their support of this work.

Finally, we thank external contributors. Chika Yoshimura helped us in evaluating the implementation. Robert Watson told us details about FreeBSD's SMP support and also incorporated our proposed fix to eliminate an unnecessary kernel lock. George Neville-Neil and Rick Jones carefully read a draft of this paper and provided valuable comments.

References

[1] Anderson, J. H., Kim, Y.-J., and Herman, T. Shared-memory mutual exclusion: major research trends since 1986. *Distributed Computing 16*, 2-3 (September 2003), 75–110.

[2] Arends, R., Austein, R., Larson, M., Massey, D., and Rose, S. DNS Security Introduction and Requirements. RFC 4033, March 2005.

[3] Berger, E. D., McKinley, K. S., Blumofe, R. D., and Wilson, P. R. Hoard: A scalable memory allocator for multithreaded applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)* (November 2000), pp. 117–128.

[4] IEEE. Information Technology – Portable Operating System Interface (POSIX) – 1003.1c pthreads, 1995.

[5] Intel Corporation. The IA-32 Intel(R) Architecture Software Developer's Manual. Available from http://www.intel.com/.

[6] Internet Systems Consortium. ISC home page. http://www.isc.org/.

[7] Jinmei, T., and Vixie, P. Practical Approaches for High Performance DNS Server Implementation with Multiple Threads. The Seventh Workshop on Internet Technology (WIT2005).

[8] Kleiman, S., Shah, D., and Smaalders, B. *Programming With Threads*. Prentice Hall, 1996.

[9] Mellor-Crummey, J. M., and Scott, M. L. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming* (April 1991), pp. 106–113.

[10] Mockapetris, P. DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION. RFC 1035, November 1987.

[11] NLnet Labs. NSD home page. http://www.nlnetlabs.nl/nsd/.

[12] Nominum, Inc. How to Measure the Performance of a Caching DNS Server. Available from http://www.nominum.com/.

[13] SUN Microsystems. SPARC Assembly Language Reference Manual. Available from http://docs.sun.com/.

[14] The Apache Software Foundation. Apache HTTP server home page. http://httpd.apache.org/.

[15] The Open Group. The Single UNIX Specification, Version 2, System Interface & Headers (XSH), Issue 5, 1997.

[16] Vixie, P., Thomson, S., Rekhter, Y., and Bound, J. Dynamic Updates in the Domain Name System (DNS UPDATE). RFC 2136, April 1997.

[17] Watson, R. Introduction to Multithreading and Multiprocessing in the FreeBSD SMPng Network Stack. In *EuroBSDCon 2005*.

Notes

[1]We originally began this work with a different OS and machine architecture, but we showed newer results for consistency with the evaluation environment described in later sections.

[2]The version we used is described in the web page of authors of [9].

[3]We did this test with Solaris 10, which was the only platform we successfully linked the Hoard library to BIND9.