Implementation and Evaluation of Moderate Parallelism in the BIND9 DNS Server

JINMEI, Tatuya / Toshiba
Paul Vixie / Internet Systems Consortium

[Supported by SCOPE of the Ministry of Internal Affairs and Communications, Japan.]

June 2006 | Usenix Annual Technical Conference

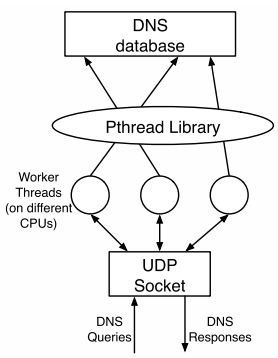
Contents

Background

- BIND9: widely used DNS server implementation
  - Richer functionality: DNS Security, better support for IPv6
- BIND9's problem: poor performance
  - thread support doesn't benefit from multiple CPUs
  - often run with threads slower than old version (BIND8)
  - => may hinder deployment of the new functionality
- Our goal: improve BIND9's response performance with threads
  - Authoritative, Caching, with or without dynamic updates
  - Performance measure: # of max queries processed w/o loss
  - Quantitive goal
    - add 50% of 1-CPU query rate for every additional CPU
    - (if BIND9 can operate 80% as fast as BIND8, it will outperform BIND8 with two CPUs)

BIND9 Implementation Architecture

- In-memory DNS database
- Worker threads (up to # of CPUs) process queries
  - Use pthread locks for thread synchronization



Profiling: measured overhead of acquiring locks

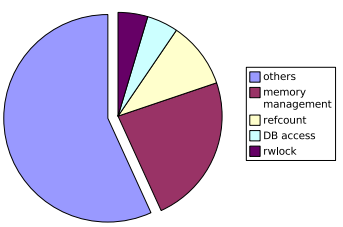
- Target
  - AMD Opteron x 4 + SuSE Linux 9.2 (kernel 2.6.8)
  - Configured as "F-root" server
- Method
  - Sent various queries, collected wait period for acquiring each lock

```
gettimeofday(t1);
pthread_mutex_lock();
gettimeofday(t2);
```

  - wait period = t2 - t1
- Analyze
  - Dumped the entire result
  - Identified dominant locks in the source code

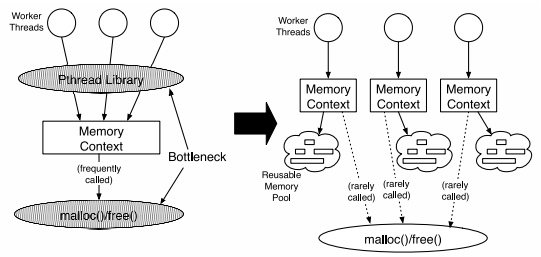
Profiling Results

- 43.3% of total running time was occupied for waiting for acquiring locks
- Of the waiting period:
  - 54.0% were for memory management in building responses
  - 24.2% were for incr/decr of reference counters
  - 11.4% were for contentions in DNS DB access
  - 10.4% were in BIND9's internal rwlock implementation



Eliminating Bottlenecks 1: memory management

- Problem: contentions in memory management for response packets
  - In a BIND9 subroutine and in the malloc()/free() libraries
- Solution
  - Enable internal memory allocator with memory pool
  - Separating work space for each thread
    - it's temporary data and doesn't have to be shared by threads



Eliminating Bottlenecks 2: operations on counters

- Problem
  - So many operations on reference counters
    - each protected by a pthread lock, causing contentions
    - => operation is pretty simple: incrementing/decrementing integers
- Solution
  - Atomic operations without locks
  - Using dedicated HW instruction or other primitives + busy loop
    - x86/AMD: xadd instruction
    - Sparc/Itanium: compare-and-swap(CAS) + busy loop
    - PowerPC/Alpha: locked load + store conditional + busy loop

Eliminating Bottlenecks 3: efficient rwlock for DB access

- Problem: lock contentions in DNS DB access
  - Even though it's read-only in most cases
  - Why didn't rwlock help?
    - 1. cannot use it due to write operations on reference counters
    - 2. custom version of rwlock (for fairness) that depends on pthread locks
- Solution
  - Implementing more efficient rwlocks
    - use rwlocks wherever appropriate
    - in a more effective way (next slide)
  - Based on Mellor-Crummy's algorithm
    - use some atomic ops on a 32-bit integer
    - make concurrent readers run fast
    - ensure fairness using pthread locks (should be rare)
  - Using dedicated HW instructions or other primitives + busy loop
    - e.g., x86/AMD: xadd/cmpxchg instructions

Efficient Rwlock + Atomic Counter Ops

- Original Implementation

```
pthread_mutex_lock(data->lock); /* may block */
data->refcount++;
value = data->value;
pthread_mutex_unlock(data->lock);
```

- New Implementation

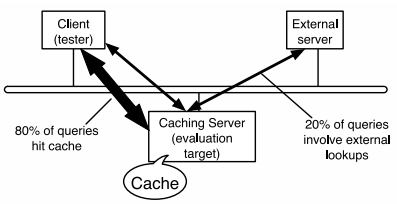
```
atomic_add(data->refcount); /* fast */
read_lock(data->lock); /* usually fast */
value = data->value;
read_unlock(data->lock);
```

Evaluation

- Hardware/Software
  - AMD opteron 2GHz x 4, RAM 3.5GB
  - Broadcom BCM5704C Dual GbE
  - SuSE Linux 9.2(64bit)
    - kernel 2.6.8, glibc 2.3.3
  - BIND9(unpublished, to be 9.4), BIND 8.3.7
- Server configurations
  - Emulated "F-root" server (as of October 2005)
  - Caching server
  - Large scale servers
  - "Dynamic" server
- Evaluation procedure
  - Sent queries from external machines
    - measured max query rate responded without loss
    - using BIND9 queryperf

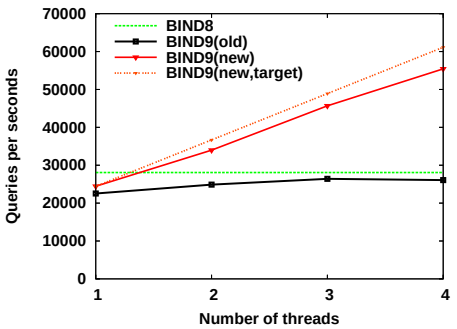
Evaluation Query Details

- For the root configuration
  - used real query data to F-root (as of October 2005)
    - 22.9% of queries were names under .BR
    - < 50% of queries were names under top 6 domains
    - => should cause contentions in DB access
- For the cache configuration
  - controlled cache hit rates with another external server
  - concentrated on the case with 80% hits
    - (number from statistics of an existing busy caching server)



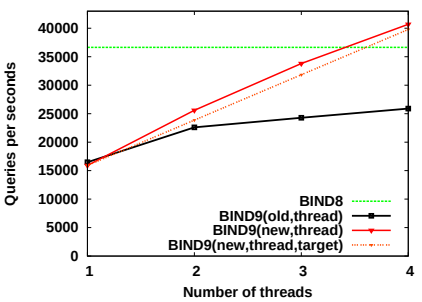
Evaluation results (root)

- BIND9(new): proportional to # of threads
  - outperform BIND8 with 2 or more threads
- BIND9(old): does not benefit from multiple threads
  - even worse than BIND8 with all available threads



Evaluation results (cache)

- Generally scaled well
  - meet our quantitive goal
- Yet not fully satisfactory
  - needed all 4 threads to outperform BIND8
  - due to lower base performance (w/ single thread)



Other Results

- Dynamic / large scale server performance
  - generally scaled well
- Memory footprint
  - even smaller thanks to the internal allocator
- Other scalable memory allocator (Hoard)
  - didn't see much difference, but we need more experiments with a larger number of CPUs
- Rwlock performance variation
  - vary among OSes
- Found and fixed FreeBSD kernel bottleneck due to unnecessary lock
  - will appear in FreeBSD 7.0

Conclusion / Future Work

- Improved BIND9's response performance with multiple threads
  - Identified and eliminated synchronization overhead
  - Confirmed the effect with a 4-way machine
  - Should be applicable to other thread-based applications
- Future work
  - Get feedback, improve implementation
    - now available as 9.4.0a5, being tested
  - Further evaluation
    - for a caching server with actual query pattern
    - other OSes, machine architectures
    - with a larger number of CPUs
    - effect of scalable memory allocator (e.g. Hoard)