Rsyslog: going up from 40K messages per second to 250K

Rainer Gerhards

What's in it for you?

- Bad news: will not teach you to make your kernel component five times faster
- Perspective
  - user-space application
  - going from single core to multi core
- Lessons learned
  - things we got wrong
  - how we improved the situation
- our experiences hopefully useful for other user-space applications as well

So what is rsyslog?

- modern syslog message processor
- Forked from sysklogd
  - Some initial coding started 2003
  - Single-threaded design and pretty old code
  - But it worked!
- Really got momentum when Fedora looked for a new syslogd in 2007
- has become the de-facto standard on most distributions

Rsyslog project…

- Design goals – around 2004
  - Drop-in replacement for sysklogd
  - Easy to use for simple cases
  - Powerful for complex cases
  - High performance and support for tomorrows multi-core machines
- Very heavy hacking in 2007 and 2008
  - Many, many features added
  - No time to consolidate them

How does rsyslog relate to other apps?

- Rsyslog actually is
  - a *message router*
  - processing mostly *independend*
  - *somewhat similar*
  - *objects*
  - within a type of *pipeline.*
- This makes rsyslog, and its problems, similar to many other (server) applications.

Performance Optimization Project

- rsysog deployed in high demanding data centers
- early v4
  - could handle 40K mps
  - scaled very badly on multiple cores
- Project goals
  - speedup processing of single message
  - improve scalability
- phase one – winter/spring 2009
  - focus of this talk
  - resulted in up to 250K mps as reported by some users

Classes of Optimizations

- Traditional optimizations
- Refactoring
- Memory-subsystem based optimizations
- Concurrency-related optimizations

Traditional Optimizations

- The boring stuff, still useful to look at…
- C strings vs. Counted Strings
- Operating System Calls
  - beware of context switches!
- Buffer Sizes
- More Specific Algorithms
  - don‘t let seldom-used features constrain often-used ones
  - use specific (fast) code for common cases

Code Refactoring

- “time to deliver” was initially dominant
- few external reviewers
- own review, found lots to change, e.g.
  - Unnecessary parameter formatting due to “interface” changes
  - Unnecessarily deep function nesting due to functionality being shuffled between functions

Refactoring: Design Review

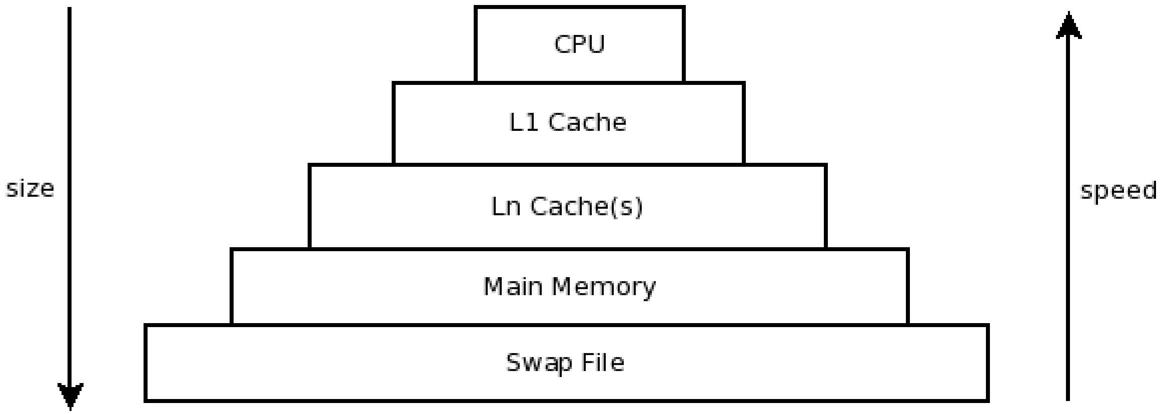
- e.g. the “no worker” really dumb case…
  - worker pool management was very complex
  - core design failure: we thought it would be useful to stop all workers when no work was done
  - of course, that was wrong:
    - keeping one blocking doesn’t require many resources
    - but restarting one does!
  - We removed that capability and got faster and easier to maintain code with less bug potential
- More potential for this kind of refactoring, e.g. (over-engineered) network driver layer

Memory-Subsystem: old ideas

- Access to memory is often considered equally fast
  - to all memory locations
  - for both reads and writes
  - this builds the basis for (almost?) all academic reasonings on algorithm performance
- It often is assumed that aligned memory access is **always** faster than unaligned access

Memory Subsystem: today‘s reality

- Access time is **very different** depending on which memory is to access and when
- Writes are **much** slower than reads
- Unaligned access may be faster for some uses



Memory: important concepts

- locality
  - spatial
  - temporal
- working set
  - minimum amount of memory needed to carry out a closely related set of activities
  - for rsyslog: memory needed to receive, filter and output a message
- goal is to achieve spatial and temporal locality for the working set!

Memory: malloc subsystem

- try to reduce number of malloc calls
- malloc instead of calloc
- using stack instead of heap where possible (but makes memory debugging much more difficult)
- (somewhat) larger malloc‘s are OK
- fixed buffers instead of malloc
  - use common size for fixed alloc inside structure
  - malloc only if actual size is larger
  - great for small elements (< 8Byte ⇔ ptr size!)

Memory: keep related things together

- Fixed buffers (as shown on last slide)
- Structure packing
- Use bit fields where appropriate (but only then)
- but move unrelated things away from each other
  - when written to by different threads (counters!)
  - otherwise cache thrashing may severely affect performance

Memory: reuse memory regions

- improved buffer management to make it most likely that a memory region is continously being accessed by the same thread
- „properties“
  - objects that keep their value for a relativly long time (many messages)
  - allocated and written once, read (very) often
  - reference counted

Concurrency

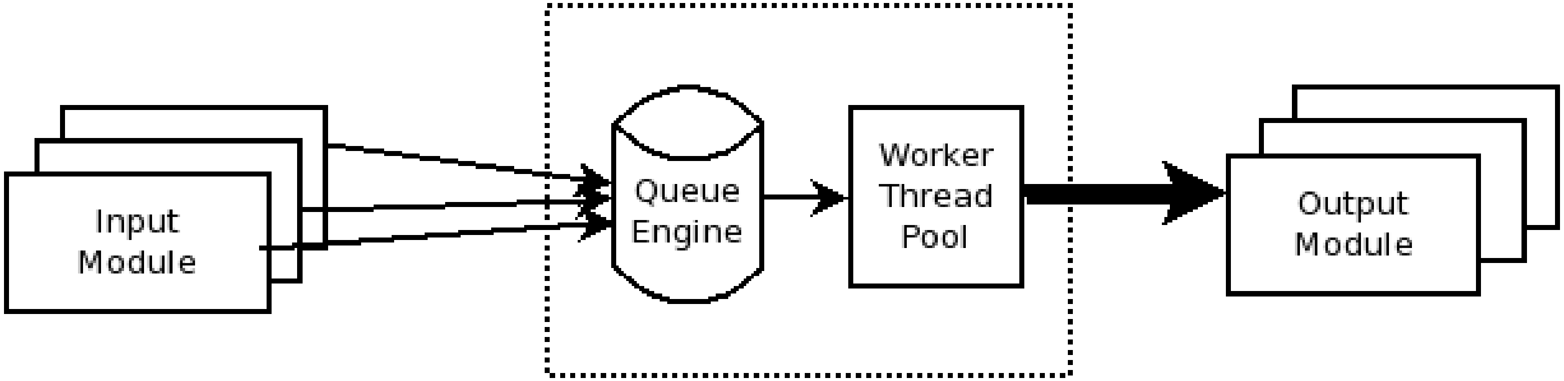
- paradigm shift: software must exploit concurrency directly, single core does not get much faster
- rsyslog started deploying multi-threading very early, with some (dumb, again ;-)) mistakes made

Problem seen in Practice

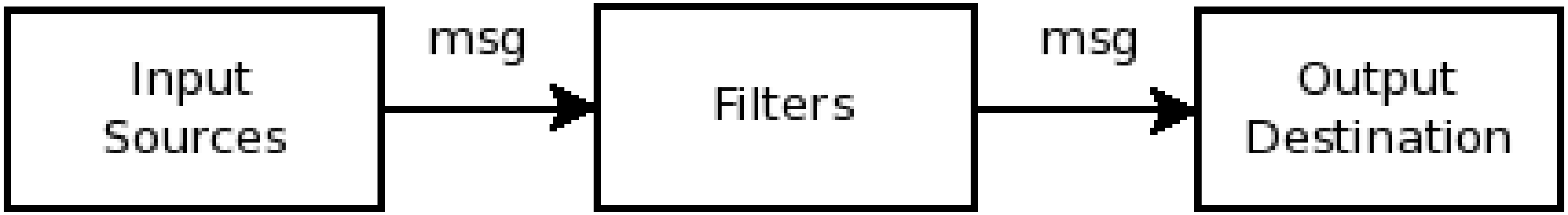
- Lock contention limited performance
- and decreased performance
  - when adding additional threads
  - with fast output processing
- because
  - lock contention dramatically increased
  - locks then needed to go to kernel space, what became the dominating performance factor

Rsyslog Design (rough sketch)

- Concurreny:
  - Each Input
  - Queue Workers
  - Output Modules (potentially)



Classical User Perception of syslog



- sequential
- assumes that sequence of messages in log store equals sequence of events

Root Cause: Usual Assumptions are invalid!

- storage sequence does not reflect event sequence
  - buffering due to unavailble target system
  - interim systems (including network reordering)
  - multithreading on any sender or receiver
  - scheduling order
- in short: **sequence can only be preserved in a toally sequential system**, which we do not have (and do not want!)

So, what‘s the solution to Sequence?

- use a „kind of timestamp“ / order relation
  - high-precision timestamps inside messages
  - timestamps with sequence numbers
  - Lamport Clocks (no implementation so far)
- then, process logs according to the selected order relation
- bottom line: sequence does not need to be preserved at the syslogd level, because **it cannot do so**!

How this affects rsyslog…

- **single most important fact** in respect to rsyslog design and performance
  - rsyslog‘s initial design tried to preserve message order as much as possible
  - severely blocked partitioning of workload
- performance optimization gained benefits from this insight
  - now, almost everything could be done highly concurrent!
  - (most) often invisible to user
  - users who don‘t like it, can turn it off

Workload Partitioning

- process messages in batches of many instead of individually
  - reduces number of mutex calls dramatically
  - reduces lock contention even more (less likely)
  - positive side effects an other items as well
  - kind of „temporal partitioning“
- multiple „main“ message queues
  - inputs can submit messages to defined queues
  - totally independent queues
  - no locking contention at all between queues

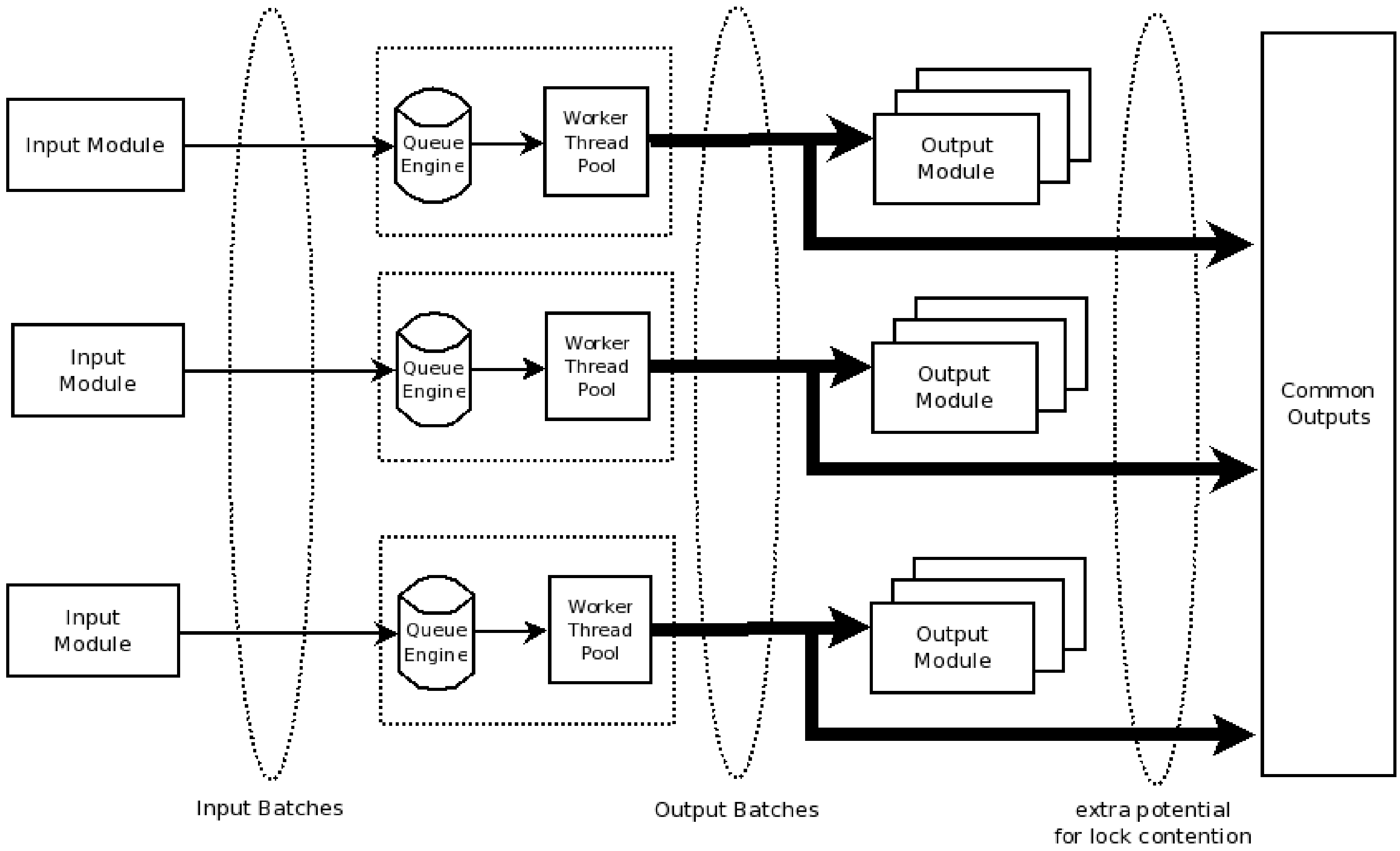
Locking Improvements

- Simplified locking primitives
  - removed need for recursive mutexes
  - evaluated code and selected fastest locking method that did the job
- Atomic operations
  - replaced locking for simple cases (counters)
  - will become more important when lock/wait-freedom is addressed in third tuning effort winter 2010/11

Some other Things

- moved functionality to different pipeline stages
  - utilizing different levels of concurrency
  - example: message parsing from input stage to main queue worker thread
- reduce hidden looks
  - some subsystems guard operations by locking
  - calling them thus serializes processing
  - sample: malloc subsystem, other libraries as well

Architecture after Redesign



Are we done now?

- no, definitely not
  - still scales far from linear for large number of cores
- second tuning effort done in spring 2010
  - brought another speedup of four
  - focussed on common use cases
  - first „exploration“ of lock-free algorithms
- third effort planned for winter/spring 2010/11
  - primary focus will be lock-freedom
  - hopefully will come close to near-linear speedup

Conclusion

- We often needed to look at a very fine-grained level to achive high-level improvements
  - We did some of the usual stuff,
  - refactored some anomalies of a fast growing project,
  - took a close look at modern hardware,
  - but **most importantly needed to break with traditional perception**.

Most important lesson learned

Re-evaluating current practice and questioning old habits is probably a key ingredient of moving from the mostly sequential programming paradigm to the fully concurrent one demanded by current and future hardware.

Many thanks for your attention

- Questions?

- rgerhards@hq.adiscon.com