Software Performance & Scalability: A Cheatsheet

by Mike Kramlich

revised 2018 January 15

1. Do Nothing (why? math-wise is perfect: min possible latency, cpu, mem & infinitely scalable)
2. Do Very Little (why? broadstrokes is the next most perfect & efficient thing after Do Nothing)
3. Static Not Dynamic
4. Cached
5. Distributed
6. Parallelized
7. Asynchronous
8. Incremental
9. Step Minimization
10. Paginated Results
11. Complexity (in Time or Space) Cost Optimal Algorithms (eg. O(1) over O(n) over O(n^2))
12. Event-Driven Not Polled
13. Non-Blocking IO
14. Web Page Component Request Minimization
15. Network Locality (eg. CDN's)
16. Machine Task-to-Data Locality (eg. Hadoop)
17. Precompute Predicted Requests
18. Eager Init vs Lazy Init
19. Higher CPU Clock Speed
20. Higher CPU Core Count
21. CPU Instructions Which Do More Work Per Cycle/Tock (eg. SIMD)
22. Higher Communication Bus Speed, Throughput
23. Do Tasks in Hardware Rather than Software
24. Leaner Languages & Runtimes
25. More Memory
26. Faster Memory
27. More Disk/Storage
28. Faster Disk/Storage (Seek, Read, Write)
29. Disk/Storage Defragmentation
30. Local Disk/Storage Rather Than Network Mounted
31. Higher Network Bandwidth
32. Compression of Large Transfer Payloads
33. Persistent Connections
34. Connection Pools
35. UDP not TCP
36. Minimize Chattiness of Comm Protocols
37. Pass Smaller Messages
38. Make Use of Otherwise Unused Local GPU For General/Parallel Compute Tasks
39. Make Use of Cloud Computing Services (eg. AWS)
40. Tuning OS Parameters
41. Custom OS Kernel Builds
42. Non-Virtual OS Instance
43. RTOS (if require a guarantee & hard upper bound on latency of action in response to event)
44. No OS (yes extreme but consider case of micro-controller where only 1 master program req)
45. Pass and Store Diffs Rather Than Complete Snapshots
46. Client-Server Architecture (eg. benefit: long startup init tasks done in server not clients)
47. Push Work To Client-Side Rather Than Server-Side (eg. rendering, initial input validation)

48. Local Function Calls Rather Than RPC or Web Service Requests
49. Function Memoization
50. Function Inlining
51. Loop Unrolling
52. Less Unnecessary Call Descent Depth (eg. Java/Enterprise Design Pattern Astronaut Arch)
53. Less Memory Churn and Background GC Inside Your Process
54. Object Pooling
55. Clusters
56. Queues with Worker Process/Thread Pools
57. Database Indexing
58. Database Stored Procedures
59. Database Prepared Statements
60. Database Sharding
61. Old Data Warehousing/Archiving
62. Old Metric/Event Rollups
63. Log Archiving
64. Timeout Guards
65. Buffer Size Tuning
66. Queue Size Tuning
67. Timeout Duration Tuning
68. Network Packet Size Tuning
69. Disk Page Size Tuning
70. Memory Page Size Tuning
71. Cache Size Tuning
72. Cache Eviction/Expiration Policy Tuning
73. Page Boundary Alignment Optimization
74. Powers of Two (because powers of 10, 3, 8, 7, 12, etc are far less efficient, more wasted)
75. Bit Packing (making every bit yield the maximum signal/use for the buck)
76. Run Bottleneck Processes with Max Priority, Minimum Niceness
77. Avoid Need To Mitigate Risk Of Hardware Failures Due To Vibration/Shock
78. Avoid Need To Mitigate Impact Of Environmental Radiation (eg. cosmic x-rays)
79. Reduce Physical Volume (eg. consider design impact on smartphones and data centers)
80. Reduce Physical Mass (eg. consider compute capabilities & cost impact on space payloads)
81. Reduce Power Consumption (eg. smartphones, laptop battery life, data centers)
82. Reduce Heat Emission (eg. impacts cooling reqs thus total cost/complexity of data centers)
83. Keep Hardware Cool, Especially Processors
84. Increase MTBF of Least Reliable Hardware Component
85. Commodity Priced Hardware Rather Than Vendor Monopoly/Patented/Unique Hardware
86. Later/Recent Generation Hardware Models (in general: more optimized than earlier/older)
87. Decrease Max/Mean/Min Time Before Detection of Failure
88. Upstream DoS Throttling/Filtering
89. User Request Throttling
90. Automatic Load Balancing (esp smarter)
91. Off-Peak Scheduling of Tasks When Possible
92. Encourage/Require Users To Spread Access/Requests/Workloads More Evenly Over Time
93. No Encryption
94. Minimize Core/Thread Context Switching
95. Avoid Mem/Disk Paging/Swapping, Especially Thrashing
96. Higher OS Scheduling Priority For Processes Directly Responding To Live User Commands
97. Avoid Lock Contention
98. Textual UI's and CLI's Rather Than GUI's
99. Text Rather Than Images and Static Images Rather Than Video, Audio or Animations
100. Vector Illus/Anims/UI's Rather Than Bitmaps To Minimize Disk/Net/Mem Footprint
101. JSON/CSV/etc Rather Than XML (whose impls often cause higher latency or mem use)
102. Prefer Precise, Reactive, Resource Sipping Tools (eg. lat/cpu/mem of vim over Eclipse)

103. Automated Rather Than Manual (eg. no content staff approval queue, just filter and flag)
104. Mass Viral Parasitic Disguised Idle Compute (eg. malware)
105. Humans For Tasks Where Cheaper, Faster, Most Accurate, Most Believable, Multi-Purpose
106. Make Light Speed in Vacuum Your Only Remaining Latency Bottleneck Where Possible

This document is a personal cheatsheet list of rules-of-thumb, impact factors, ideas, patterns, strategies and techniques which can be used to improve, or at least to sculpt the performance or scalability of computing systems. Whether measured overall and universally applicable, or, merely temporarily apparent from the standpoint of any particular user, viewer or stakeholder, or as measured by one particular goal metric — possibly at the expense of worsening others.

It is not a reference manual or textbook. The traditional purpose of a cheatsheet is to remind a reader of topics they have already studied and thus should know. And to cram as much information as possible onto the smallest amount of paper.

Some of the items in this list contribute also to system availability, correctness, cost (esp lifetime TCO) or the intangible quality of the user experience. Often the effects overlap and the lines blur. As with any element in software engineering there are trade-offs to consider. For example, sometimes its better to init eagerly and sometimes init lazily. Prediction-based optimizations can help or hurt: they can help at a coarse-grained level overall yet also hurt in fine-grained individual cases. Caching can shrink latency but risks showing stale or inconsistent results. And it always helps to prioritize based on actual bottlenecks. Takes judgment to know why or how to apply any of these.

*Note that some items may not at first appear to impact performance or scalability, at least not directly, but if you zoom out a bit in your mind and consider their downstream impacts or their impacts on the overall solution, they do. For example, radiation and vibration don't necessarily *directly* hurt latency or workload capacity, but *can* cause hardware failure or data corruption, which in turn will hurt your system's correctness, availability, latency, throughput, and service lifetime. Even having features present in a design in order to *mitigate* these risks can then in turn penalize performance, however small: think about synchronization, redundant writes or error correction. Or place a drag on scaling: think about increased hardware costs to serve any given level of traffic/workload or data size, or the increased cognitive burden on engineers of having to support more complex architectures, or the increased mass and volume required to add shielding or shock absorption. Think about the need to have more or better — and therefore also more rare and expensive — engineers to design/build/support such a system. Think about bang per buck. Everything is connected.*

Also note many of the items overlap partially with one another, or can be said to be more specialized cases of other items in the list (eg. function memoization is a special case of caching.) Even in those cases there is a distinction noteworthy enough to warrant separate treatment here. Tried to position related items near each other.

Feel free to suggest changes or additions!
[TODO: hierarchy, tradeoffs, visualizations, examples, tools, references]

Contact author: groglogic@gmail.com

https://synisma.neocities.org/resume_Mike_Kramlich__Software_Engineer.pdf

Feedback thanks: Neil Gunther (PerfDynamics, Voyager/Galileo), Alan Robertson (Assimilation, HA Linux, Bell Labs), Avery Pennarun (apenwarr/redo, Google Fiber), Taylor Deehan (MLB)