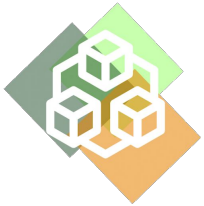
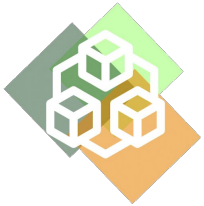


Johny’s Software Lab

My program was running fast six months ago. What happened?

Why do programs tend to get slower over time?

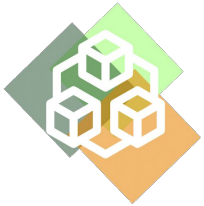
https://johnysswlab.com, @johnysswlab ivica@johnysswlab.com



About me

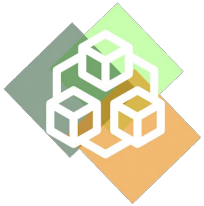
- Ivica Bogosavljevic - application performance specialist
- Professional focus is application performance improvement - techniques used to make your C/C++ program run faster:
  - Better algorithms,
  - Better exploiting the underlying hardware
  - Better usage of the standard library
  - Better usage programming language
  - Better usage of the operating system.
- Work as a performance consultant
  - Help with debugging performance issues in software
  - Trainings for teams developing performance sensitive software

https://johnysswlab.com, @johnysswlab ivica@johnysswlab.com



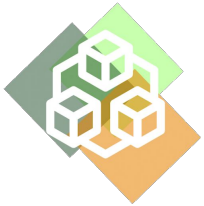
Introduction

- As software develops, more features are added to it, it seems to get slower
- Reasons:
  - Architectural issues
  - Algorithmic issues
  - Memory allocation
  - Compiler optimizations
  - Hardware issues



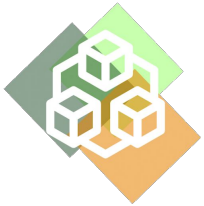
Johny’s Software Lab

Architectural Issues



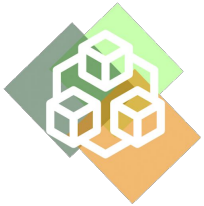
Architectural Issues

- Architectural issues:
  - API design
  - Internal component design
  - How different components work together to achieve a goal
- They can require a lot of rewrite to improve
- Careful design is important to avoid performance issues related to architecture
- For performance-sensitive systems, performance considerations need to be taken into account from day 1



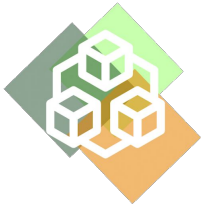
“Chatty” components

- Software consists of logical components or modules
- Components exchange information among themselves to get the work done
- API design for performance
  - Minimize the number of times component A has to communicate with component B
  - Minimize the size of the message exchanged between A and B
- Components that talk a lot are often called “chatty”
  - Processing data one by one vs processing data in bulks



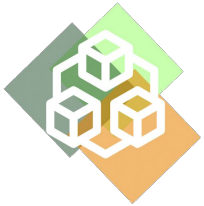
“Chatty” components

- “Chattiness” kills performance for several reasons:
  - Overhead of function calls
  - Inhibits compiler optimizations
  - Overhead of critical section protection
  - Instruction cache misses
- The price becomes much higher if components need to be moved to separate processes or separate computers
- Additional problems:
  - A component can “organize” its data better if the amount of data is known in advance
- Fixing later difficult
  - Large rewrite needed



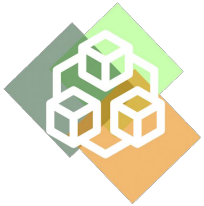
“Chatty” components - malloc example

- Memory allocation functions
  - void* malloc(int size) - allocates one block of memory of a given size
  - free(void* p) - releases one block of memory of a given size
- A program that needs to allocate 1 million objects
  - 1 million calls to malloc
- Alternative: malloc that can allocate variable number of blocks
  - block_ptr* malloc(int size, int count)
  - free(block_ptr*)
- Performance benefits:
  - The allocator can organize the memory better if it knows it will give out 1 M blocks
  - Decreased overhead of function call
  - Decreased overhead of multithreading synchronization
  - Improved instruction cache use (more on this a bit later)



Data Copying - Data Conversions

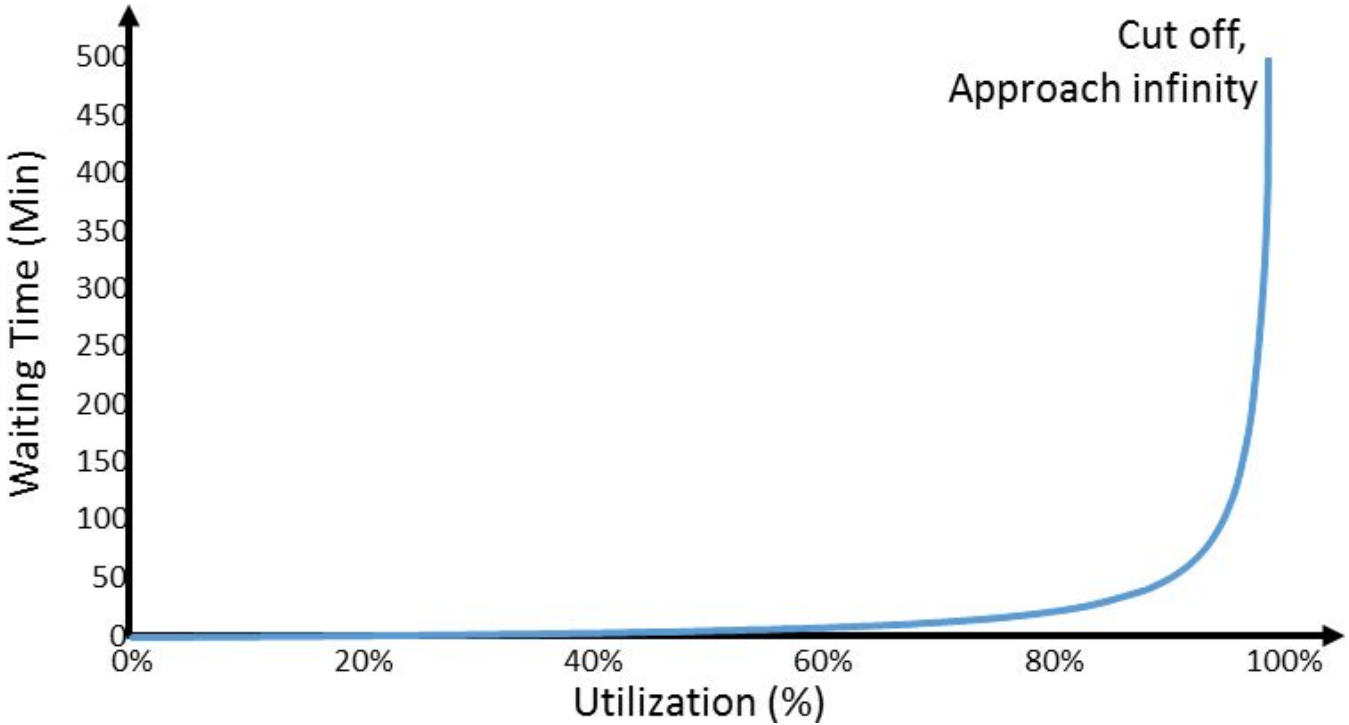
- Data copying and data conversions
  - They tend to appear more and more as the complexity of the system grows
- Neither data copying nor data conversions do any useful work
  - Should be avoided whenever possible
- Example unnecessary data copying
  - Components allocating memory for inputs or outputs
- Can be avoided with clever design
  - The data format should be agreed before either of components is designed
  - If using external libraries or components, use the data format expected by the external libraries

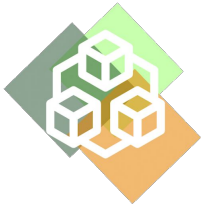


Contention

- Contention - getting stuck in line and waiting for something to happen
  - Resource contention - waiting on a resource
- Component contention
  - The main reason for contention - waiting to enter critical section
- High utilization -> waiting times explode
  - Mathematical interpretation: Kingman’s formula

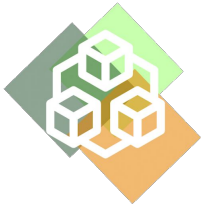
Contention


Cut off,
Approach infinity
Waiting Time (Min)
500
450
400
350
300
250
200
150
100
50
0
0%
20%
40%
60%
80%
100%
Utilization (%)



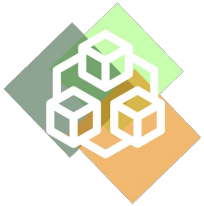
Contention - example logger

- Logger - shared component - writes logging data to a file
- Critical section protected by a mutex
- Large utilization -> many components are waiting to put data in logger
  - System becomes much slower
  - Domino effect: components that are waiting for components that are waiting for components
- The effect described by Kingman’s formula seen everywhere
  - Overloaded server
  - Overloaded database
  - Overloaded bank teller
  - Overloaded highway (Autobahn)



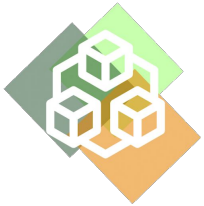
Johny’s Software Lab

Algorithmic Issues



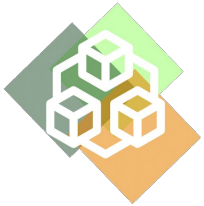
Algorithmic Issues

- Problem with data set size
- Small data set -> many loops with different complexities in performance profile
  - O(n), O(n log n), $O(n^2)$
- Large data set -> only loops with the highest data complexity in performance profile
  - The most complex loop in the program eats up all the runtime
- Programs with complexity larger than O(n log n) scale badly
  - The program runtime doesn’t grow linearly to the data set size



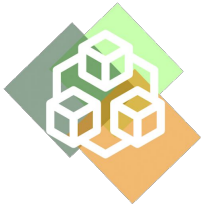
Johny’s Software Lab

Memory Allocation



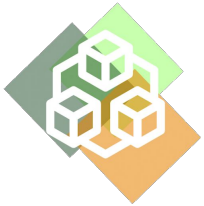
Memory Allocation

- System allocator is a shared resource
- New code (or component) that uses the system allocator a lot
  - Increases data fragmentation
  - Decreases data cache hit rate
- Result -> All components relying heavily on the system allocator get slower



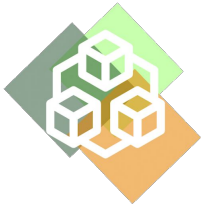
Memory Allocation

- Code that uses system allocator a lot:
  - All types of pointers
  - Binary tree based, linked list based and some hash maps based data structures
- Mitigation strategies:
  - More efficient system allocators
  - Per-component allocators

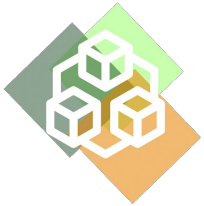


Compiler Optimizations

- Compilers optimizations are fragile
  - Compilers rely on pattern matching and heuristics
    - Vectorization and inlining
  - A code that is well optimized might get deoptimized with one additional line
  - Change of compiler or compiler version can also break optimizations
- No generic solution
  - However, writing simple easily maintainable code helps

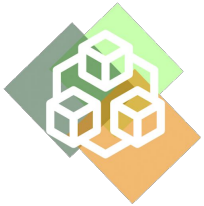


Hardware issues



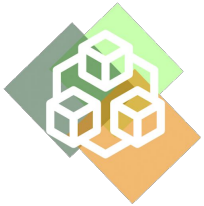
Hardware Issues

- Larger program or larger data set -> less hardware friendly
  - More instruction cache misses
  - More data cache misses
  - Failure to use CPU’s vectorization units



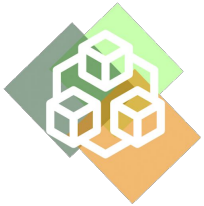
More code -> more instruction cache misses

- Instruction cache speeds up access to instructions
  - Instruction in the instruction cache -> fast access
  - Instruction not in the instruction cache -> needs to be fetches from the main memory -> slow access
  - Instruction not accessed for a long time -> evicted from the cache
- Bigger program -> more instruction cache misses
- Some programs suffer more from this issue
  - Programs that quickly move from one function to another



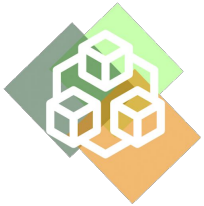
More code -> more instruction cache misses

- Mitigations:
  - BOLT
  - Profile-guided optimizations
  - Link-time optimizations



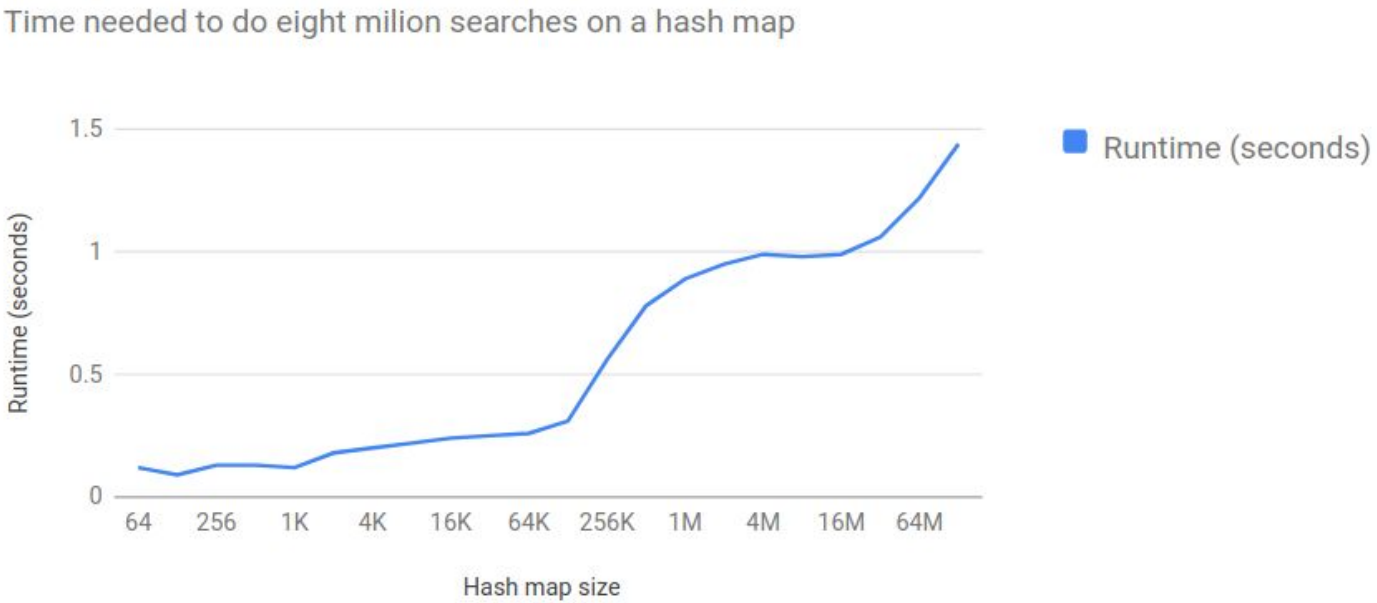
Larger workload -> more data cache misses

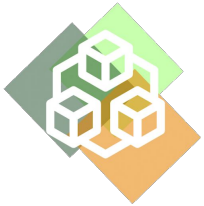
- Data cache - small memory to speed up access to commonly used dat
  - Data in the data cache -> fast access
  - Data not in the data cache -> needs to be fetched from the main memory -> slow access
- Problem with random access data structures
  - Trees, hash maps, linked lists (not arrays)



Larger workload -> more data cache misses

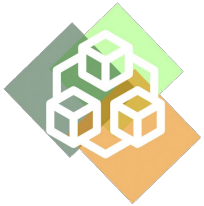
- Lookup in small data structure faster than lookup in a large one





Larger workload -> more data cache misses

- No general solution
- Mitigations:
  - Open addressing hash maps
  - N-ary trees
  - Binary trees with a good memory layout (TODO Layout)

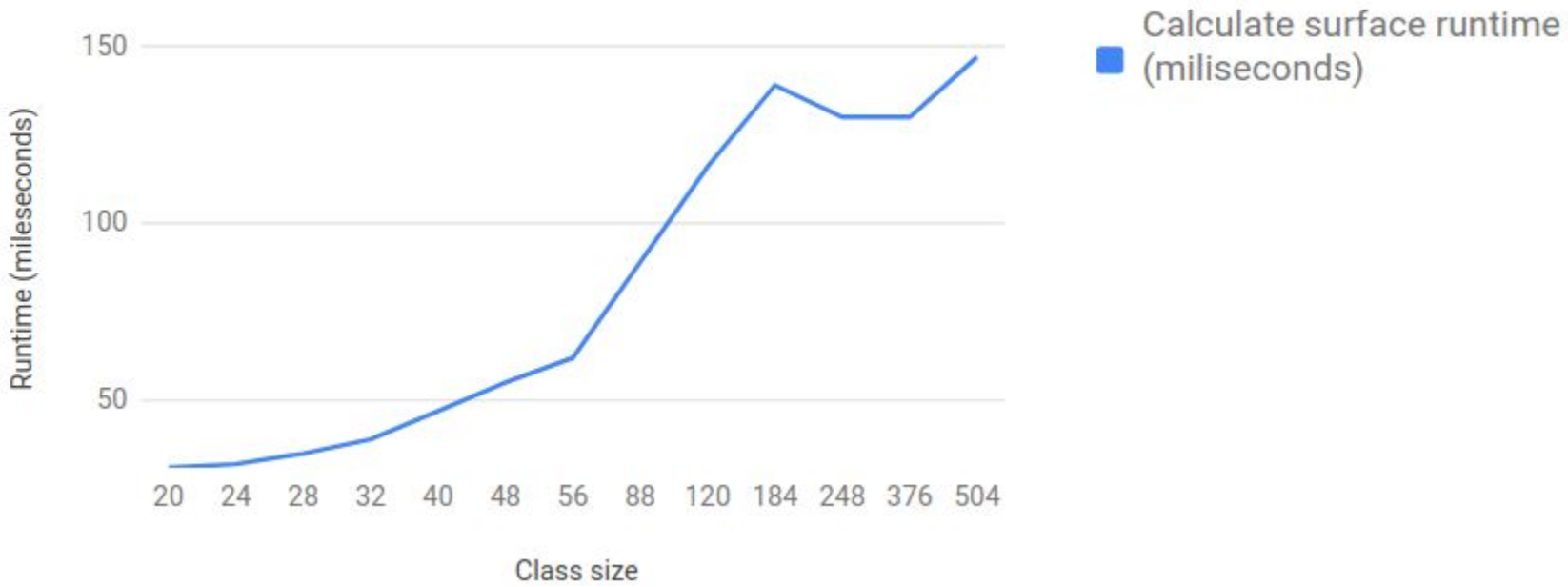


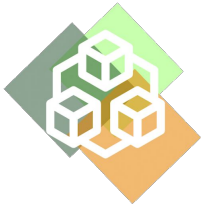
Larger classes -> more data cache misses

- Processing large classes is slower
  - Data fetched from memory to the data cache in blocks
  - Large classes bring unused data to the data cache

Larger classes -> more data cache misses

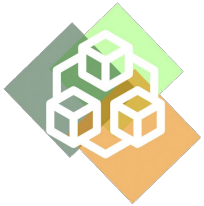
- Time needed to process 20M elements of an array





Larger classes -> more data cache misses

- Mitigations
  - Decompose large classes into smaller classes
  - Entity-Component-System



The End

- Questions?
- Interested in C/C++ software performance? Subscribe:
  - Twitter: @johnysswlab
  - Linkedin: https://www.linkedin.com/company/johnysswlab/
- Need help with performance in your program? Contact us!
  - ivica@johnysswlab.com
  - https://johnysswlab.com/consulting/