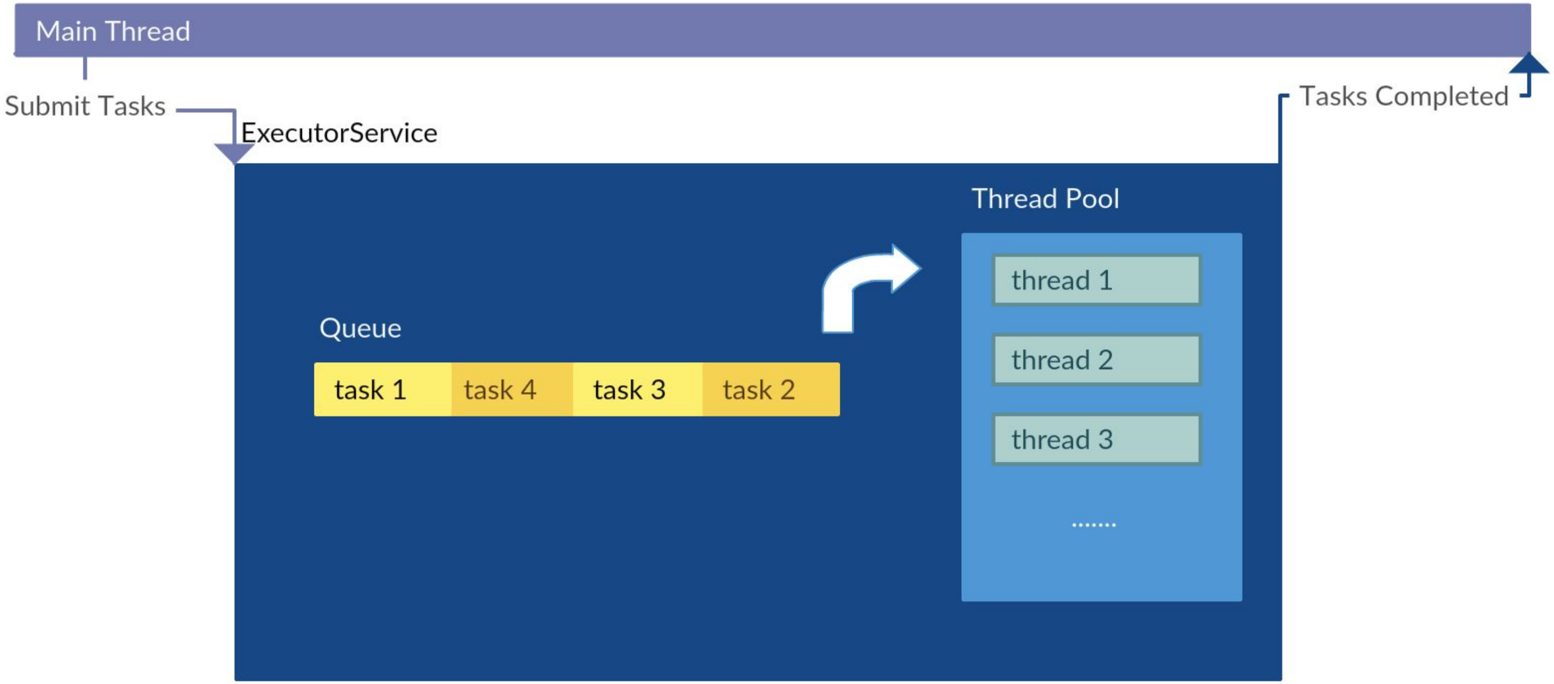
Self-adapting ExecutorService implementation

Group 21
Simon Su, Edward Zhang, Franklin Wang

Overview

- Problem Space
- Existing Implementations
- Our Ideas and Attempts
- Adapting under different conditions and Benchmarking
- GUI

ExecutorService



ExecutorService

- **Thread Creation**
- **Thread Management**
- Task submission and execution
- Shutdown

Problem Space

- How cheap are Threads?
    - Thread lifecycle overhead
    - Thread data
    - System overhead
    - Context switch
- How many threads?
    - Thread pools
    - **System load**

Comparing with Java implementations

- **ThreadPoolExecutor (Executors factory)**
    - corePoolSize
    - maximumPoolSize
    - keepAliveTime
- ForkJoinPool

Self-adaptation

Static	Dynamic
When the ExecutorService is initialized, it should be able to make a rough guess as to how many threads are suitable for the system	While tasks are being executed, the ExecutorService should adjust the number of threads in use as the overall system load fluctuates

Self-adaptation cont.

So why are we doing this?

- If we have an excessive number of (active) threads (e.g. thousands on a 8 thread system) then our performance can be impacted negatively due to context switching
- To overcome this, we should have fewer threads… however this can cause thread starvation if threads get blocked (i.e. I/O wait)
- Therefore, we *can* have lots of threads, but not so much that we have massive context switching penalties
- We should also take into account how many threads are currently blocked for I/O or similar, in order to prevent some sort of starvation

Static Analysis Ideas

- Number of available hardware threads available
  - Or number of physical cores, ignoring SMT threads
- Perform quick benchmark with various # of threads to determine optimal
  - Could “binary search”
  - Slow startup
  - Would the benchmark even reflect the actual workload? (more on this later)
  - What if system load changes while benchmark is being run

Dynamic Analysis Ideas

- Monitor real time performance metrics to determine number of threads
- System or internal task metrics
  - Internal task metrics would require instrumentation
  - Thread active time as an example of a task metric
  - Overall system CPU utilization as an example of a system metric
- Attempt to detect threads waiting on I/O to avoid starvation

Considerations

Unfortunately on real systems with other loads, MT performance can be affected by other factors we can't really measure or even control

- CPU clocks can vary based on single threaded vs multithreaded load (e.g. Intel Turbo Boost), which can affect performance unintuitively
- The type of "other" computations running on the system could affect SMT (e.g. Intel HyperThreading) efficiency

  ...especially on weird systems such as AMD Bulldozer FlexFPU - other threads using FP instructions could tank our performance, even if it doesn't affect metrics like CPU usage

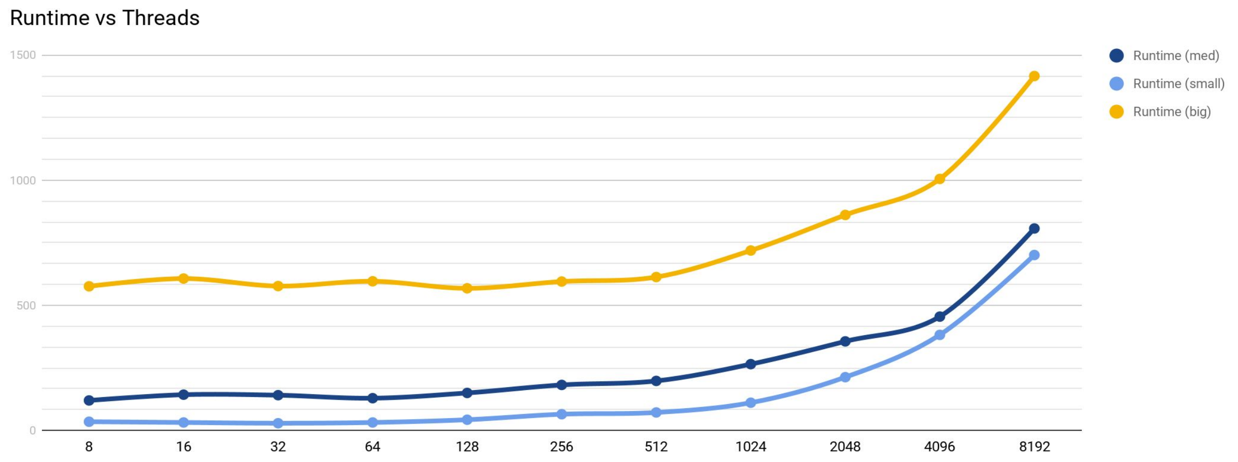
- Different sizes of work (i.e. granularity) can affect the impact of overheads
- Some cores may not be equal (e.g. ARM big.LITTLE)

However, most of this is out of scope of this project but interesting to consider nevertheless...

Static Implementation Attempt

Goal: Roughly estimate the max. number of threads we can have on the system

How: Run a quick benchmark with exponentially increasing thread numbers while measuring performance/overheads



Further work...

Can we determine max threads *dynamically*?

Can we run the benchmark *faster* when we start?

How are these values affected by *system load*?

Benchmarking

- Run separately to the JVM executing the load to simulate load
- Run different types of static tasks
  - Different problem sizes, fine <-> coarse grain
  - Efficient and inefficient for SMT/HyperThreading

Tuning under different conditions

- The ExecutorService can have a priority or “niceness” option
    - Could be a percentage of the normal number of threads used under high load
- Priority option can affect the thread count depending on system load
- E.g. low priority can lower the number of threads if system load is high such that other threads can run instead

GUI responsiveness

- Measuring responsiveness can be done by measuring the number of frames rendered by the GUI framework
  - Need to consider more than just idle performance
- Add a component to the widget tree that doesn't draw anything but measures how frequently the GUI framework invokes rendering
- ExecutorService could take in additional "factors" which act as performance metrics for determining the number of threads