Tuning Computer Systems

Valentin Dalibard

The Problem

- What is performance?
  - Resource use (time, power...)
  - Computational properties (accuracy, fairness, latency)
- How do we improve it:
  - Manual Tuning
  - Runtime autotuning
  - Static time autotuning

Outline

- Manual Tuning
  - Profiling
  - Updating the code
  - Testing performance
  - Statistical tools
- Runtime autotuning
- Static time autotuning

Manual Tuning: Profiling

- Always the first step
- Simplest case: “Poor man’s profiler”
  - Debugger + Pause
- Higher level tools
  - perf, VTune, Gprof…
- Distributed profiling: a difficult active research area
  - No clock synchronization guarantee
  - Many resources to consider
  - lprof (OSDI 2014) leverages system logs

Numbers Everyone Should Know

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

Manual Tuning: Updating the code

Two main categories:

- Change the implementation to avoid unnecessary costs
  - e.g. Make memory access pattern more local
- Tune the implementation
  - e.g. Cache eviction heuristics

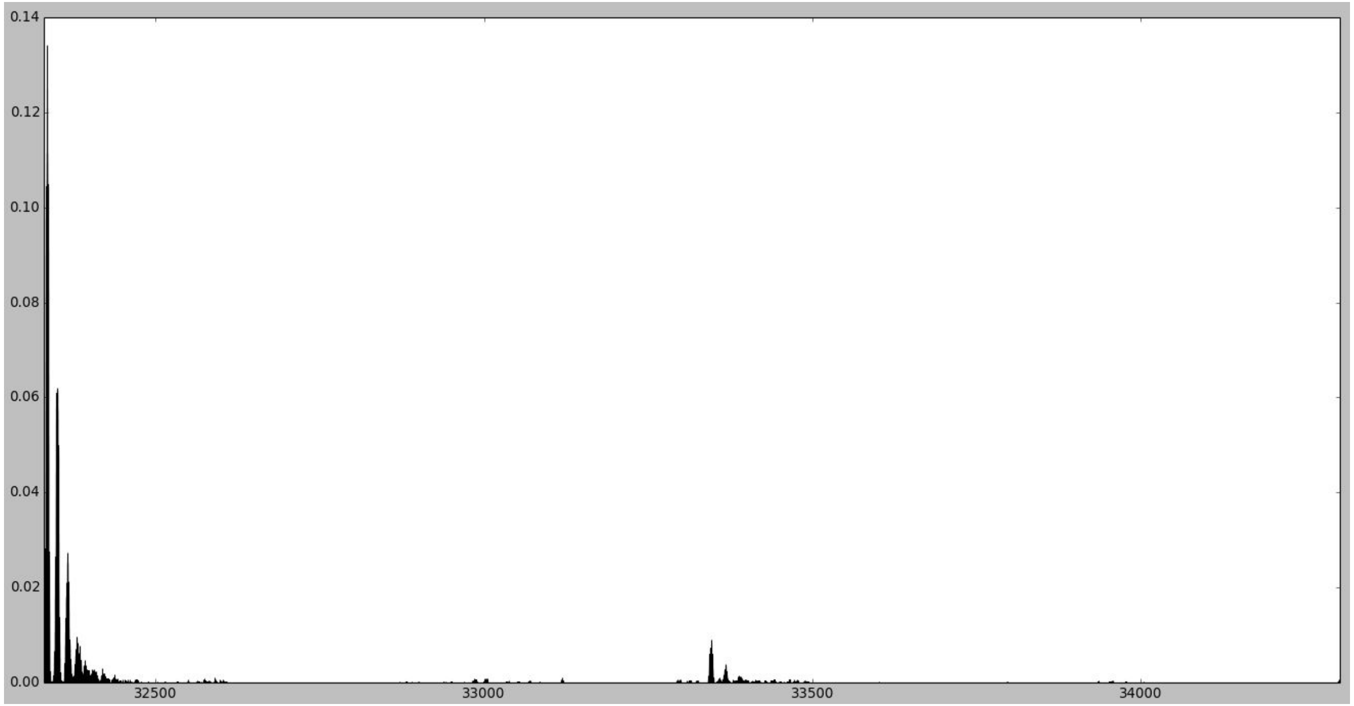
Manual Tuning: Testing performance

Slow rollout

- Benchmark inputs - Never captures all interactions
- Subset of users
- All users

Manual tuning: Statistical tools

Often impractical as real data has weird distributions



Outline

- Manual Tuning
- Runtime autotuning
- Static time autotuning

Runtime autotuning

Plug and play to respond to a changing environment

For parameters that:

- Can dynamically change
- Can leverage runtime measurements
- e.g. Locking strategy

Often grounded in control theory

Outline

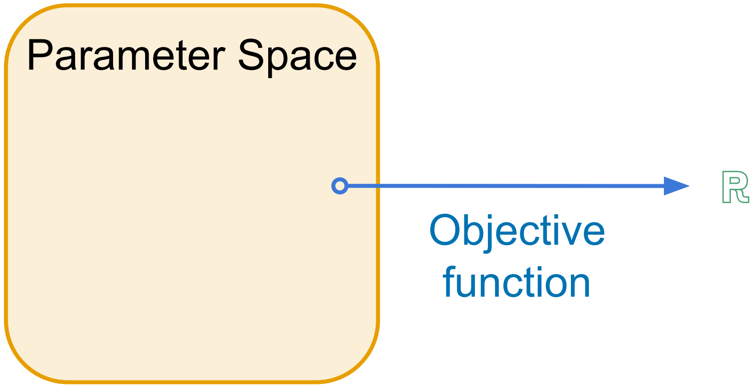
- Manual Tuning
- Runtime autotuning
- Static time autotuning
  - Phrasing the problem
  - Petabricks
  - Bayesian optimization

Static time autotuning

Especially useful when:

- There is a variety of environments (hardware, input distributions)
- The parameter space is difficult to explore manually

Static time autotuning: Phrasing the problem



Defining a parameter space

- Traditional optimization: $x \in \mathbb{R}^n$
- Suited to autotuning: Context free grammar

$$\begin{aligned} \langle sort \rangle \quad & ::= \text{insertion\_sort} \\ & \quad | \quad \text{quicksort} \\ & \quad | \quad \text{if } \langle query \rangle \text{ then } \langle sort \rangle \text{ else } \langle sort \rangle \end{aligned}$$

Petabricks: A language and Compiler for Algorithmic choice (2009)

- BNF-like language for parameter space
- Uses an evolutionary algorithm for optimization
- Applied to Sort, matrix multiplication

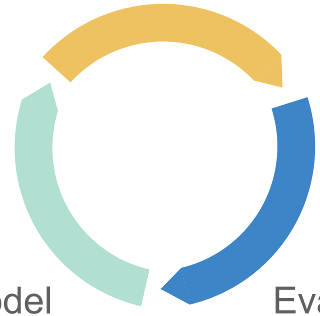
Refined in PLDI 2015 for input aware algorithmic choice

Performing the optimization can be long (hours)

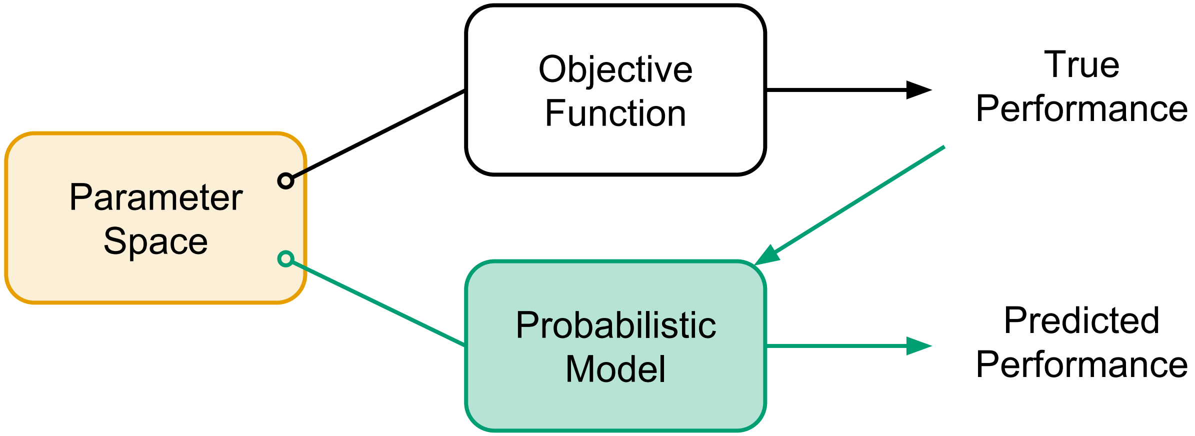
A different approach: Bayesian Optimization

For when the objective function is expensive. e.g. neural network hyperparameters

Iteratively build a probabilistic model of the objective function



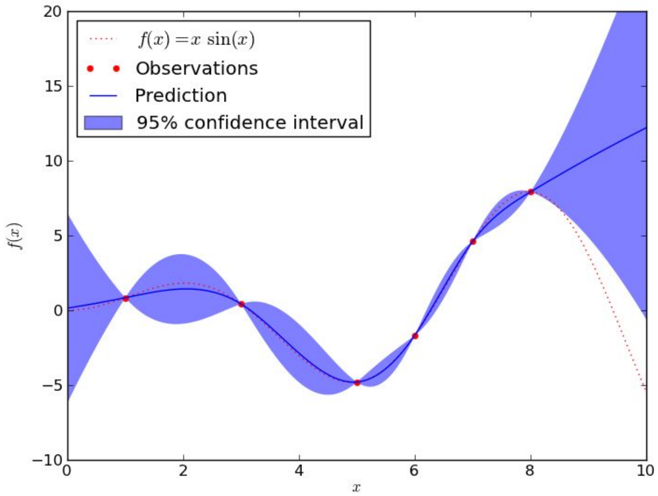
Bayesian Optimization



Probabilistic model for Bayesian optimization

Gaussian processes:

- Do regression: $\mathbb{R}^n \rightarrow \mathbb{R}$
- $O(N^3)$
- Allow for uncertainty



Acquisition function

Designed to trade-off exploration

and exploitation

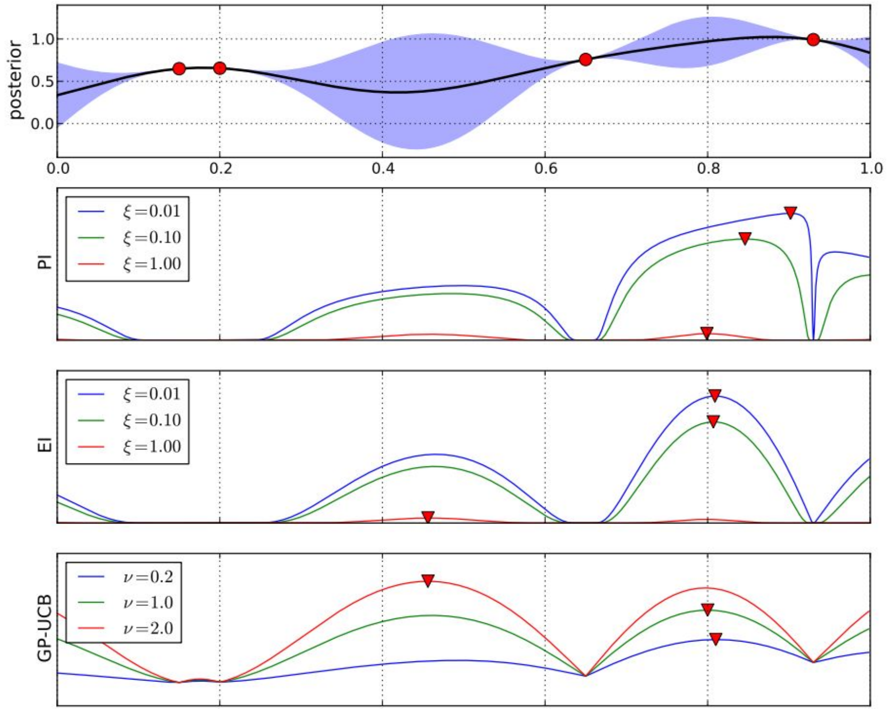


Figure 5: *Examples of acquisition functions and their settings. The GP posterior is shown at top. The other images show the acquisition functions for that GP. From the top: probability of improvement (Eqn (2)), expected improvement (Eqn (4)) and upper confidence bound (Eqn (5)). The maximum of each function is shown with a triangle marker.*

My work: Structured Bayesian Optimization

- Allow the user to add structure
- More general parameter spaces
- User given probabilistic models