Consider a programmer implementing a scheduler for their distributed system. Although the overall structure is fixed, many parameters, such as the ones in the scheduling heuristics, allow for some implementation freedom. The values of those parameters do not affect correctness, but they do impact performance and hence they must be tuned to produce an efficient implementation.

Currently, two solutions are available. On one hand, the programmer could manually tune these parameters. They would do this by leveraging some profiling tools as well as their understandings of the tradeoffs and bottlenecks involved. On top of requiring a fair amount of programmer’s effort, this approach has the disadvantage of not being robust to changes in the environment. If the network bandwidth decreases, or the input distribution changes, the tuning process will have to be executed again to account for this new context. This issue is especially relevant to computer systems which are designed to abstract an environment and hence should be performant in a wide range of settings.

On the other hand, the programmer could use a black-box optimization tool. To do this, they would expose the parameters of their scheduler to an optimizer and provide as objective function some performance metric of their system, such as the average job runtime. Recent work has shown that, when tuning program parameters, automatic optimization tools could surpass human expert-level optimizations [1]. Furthermore, using this method different environments can simply be tackled by running the optimization again.

This approach too has downsides. The black-box nature of the optimization means it can not take advantage of any information apart from the performance values observed, and hence it may be slow to converge. Practically, this means the optimization may need up to thousands of evaluation of the objective function, and will have difficulty scaling beyond dozens of parameters. This approach is also limited to scalar or categorical parameters.

In this work we extend Bayesian optimization, often used for black-box optimization, in an attempt to bridge these two approaches. We present Structured Bayesian Optimization (SBO), a new framework which lets users expose the structure of their programs to guide the optimization towards high performance parameter values.

Traditional Bayesian optimization works by incrementally building a probabilistic model of the objective function. Each iteration, a point with high performance in the model is picked and evaluated. The model is then updated with this new observation. SBO extends Bayesian optimization with two new constructs. First, rather than learning a general probabilistic model, such as a neural network or a Gaussian process, we allow users to specify structured models via our probabilistic programming framework. Probabilistic programming is a recent tool from the Machine Learning community which generalizes graphical models and makes the construction of structured probabilistic models intuitive.

This approach allows users to construct finer grained models which capture their knowledge about the behaviour of the program, essentially shifting the uncertainty from the impact of the parameters themselves, to the the tradeoffs between complex high level features such as the locality of the computation or its effect on performance. These models can decouple data properties, such as the input distribution, from environment properties, and leverage all runtime measurements to perform inference.

Second, we let users decompose the parameter space of their applications and optimize subcomponents individually according to a probabilistic utility. Every iteration, a high level optimization learns a better approximation of what utility each component should be optimizing, while low level optimizations update the parameters accordingly. This approach drastically improves scalability as it reduces the effective dimension of the optimization.

As a proof of concept, we extract the parameters of libstdc++’s std::sort() implementation. We find the optimal parameter values are highly dependent on the input’s length and how sorted it already is. Using our framework we build a decision tree parameter space which, given an input distribution and environment, generates a bespoke high performance dynamic dispatcher, trading-off query costs and quality of dispatch. Our implementation consistently surpasses std::sort()’s performance and gains orders of magnitude on some input distributions. Our current work includes the development of case studies on concurrent CPU and GPU applications, a domain which typically requires large amounts of tuning.

In conclusion, we present a compile-time framework which abstracts the values of implementation parameters in favour of the tradeoffs they optimize. Ultimately, we believe an approach like this one can help tackle the complex implementation space of statistical learning algorithms, which have flexibility in both their algorithmic and systems parameters.

References