# Generalizing a Probabilistic Auto-Tuning Method for Increased Compatibility

Alex Cannon
Swarthmore College

Brian Nadel
Swarthmore College

Aashish Srinivas
Swarthmore College

***Abstract*—Shortcomings of conventional compiler optimization, due in large part to computer architectures growing increasingly more complex, has inspired an interest in tuning algorithm-level parameters to optimize the performance of an application. While such tuning continues to be done manually, automatic tuning (auto-tuning) methods are being explored as a more efficient and effective way of configuring applications optimally. Such systems are useful for ensuring that performance-critical code can be tuned to run near-optimally on different system environments. The goal of our present project is to port a probabilistic auto-tuning system into the C programming language so that it can more easily be implemented in various software projects. We will first implement a general, lightweight auto-tuning framework which will allow users to define tuning knobs in their application and a search strategy that optimizes those tuning parameters through repeated recompilations and runs. Next, we will implement the specific probabilistic search strategy described in [2] as a plugin to this framework. If time allows, we will also test this system on an auto-tunable application and compare results from different search strategies and auto-tuning frameworks.**

## I. Background

Auto-tuning is the process of tuning program parameters to optimize some measure (often performance) on a given computer architecture. In general, the goal of an auto-tuning algorithm is to maximize the value of an objective function subject to some constraints. A naive method of auto-tuning is the hill-climbing or gradient-ascent approach. In this method, we can find a local optimum by repeatedly finding the slope of the objective function at particular points, then moving in the direction of greatest increase. The algorithm terminates upon reaching a local maximum.

However, algorithms like gradient-ascent are highly dependent on the starting set of parameters, and they generally do not make use of all the information available to them at any point. In addition, the search space for many auto-tuning problems are highly irregular and jagged. This makes gradient-ascent particularly ineffective, as there are many sub-optimal local maxima. Moreover, high-performing parameter configurations are often close to low-performing or crash-inducing configurations, as the best configurations occur when system resource utilization is pushed to its limit. The intuition of [2] is that we can use a probabilistic method to make better predictions about the performance of untested parameter configurations. Particularly, for any untested configuration c, we try to find the k-nearest neighbors n1 , , nk (where distance is the manhattan distance between ni and c in the space of tuning parameters). Then, we use these nearest neighbors to extrapolate how the system will perform under the untested configuration c. Using this method allows for a more informed search of the complex space than is possible under gradient-ascent.

## II. Motivation

One implementation of this probabalistic auto-tuning framework has yielded promising results [2] . However, its current implementation is designed within a very specific com- puter infrastructure, and its compatability is severely limited. A large part of this limitation stems from the fact that the whole package is written in Java. Not only does this limit performance, but it requires the entire Java Virtual Machine to be distributed along with the software. Other specific design choices, like storing auto-tuning output in Xcel files, also limits portability.

By rectifying some of these compatability limitations mentioned previously, we hope to create a new auto-tuning framework that draws heavily from the system outlined in [2], but that is more modular, lightweight, and generalizable. Ultimately, our hope is that this bare-bones framework will become a new standard in lightweight auto-tuning, much like SQLite has become the standard for lightweight databases.

## III. Our Idea

A large part of our project will be translating large parts of the system in [2] from Java into C. Though a fairly straighforward undertaking, this process is by no means trivial. One source of difficulty may be removing large parts of the original package from their web of Java-related dependencies. In doing so, however, a majority of the compatability restrictions will be resolved. This will allow our framework to be applied to the vast universe of C code, and also make the tool more light-weight. Another important point is that some features are much easier to measure in C as opposed to Java. For example, the source code for perf and ps are written in C. The first can be used to measure the cache hit/miss rate and other importance processor-level performance statistics while the second can be used to measure memory usage. We can bundle the source code for these applications with our software and use them to measure different performance characteristics. This will allow users to define more complex and specialized objective functions and constraints for auto-tuning. In addition, a C implementation will make significant

performance improvements as compared to its original Java implementation.

The second main part of the project will making the existing design more modular. By allowing search strategies to be easily interchanged, not only will the framework be customizable according to the wishes of future users, but we will be able to compare the performance of the probabilistic search strategy to other more naive strategies. This will allow us to better understand the relative merits and flaws of each search strategy, and to expand off of the evaluation performed in [2]. On a similar note, we will attempt to generalize smaller ad-hoc design decisions in the original implementation. Rather than directing output to Xcel spreadsheets, for example, we intend to use an SQLite database. Library of STuneLite parameters? Environment Variables? "Forking" a new process?

## IV. Related Work and Possible Directions For Future Work

A relatively flexible auto-tuning framework already exists. ActiveHarmony, the 4.5 version of which was released as recently as September 30, is an autotuning framework that supports the use of interchangeable search strategy plugins, an implementation of the same kind of modularity we hope for in our design [1]. Admittedly,the differences between Active- Harmony and the framework we seek to design our unknown, though our intuition is that our design will be more lightweight. Though time will likely not allow us a full comparison between our system and ActiveHarmony, such a comparison may be useful direction for future work.

## V. Goals for Milestones

The first goal of our project will be to developing a skeleton auto-tuning framework. For some program with a simple search space, we intend to first get a trivial search strategy working (most likely a "try every candidate" point strategy). In order to properly test, we will also need to identify such a program we can use. We may split up these tasks such that one person focuses on identifying a test program and interfacing it with our framework, (this will likely involve writing a suitable application wrapper), while the other two focus on developing the autotuning program itself. Ideally for Milestone 1, we hope to put together an autotuner that produces some tangible result. If we are unable to accomplish this, we will at least be able to present a greater understanding of the code, and some partial progress toward a working program.

For milestone 2, we hope to develop a more complete and interesting program that uses a non-trivial search strategy. This will also require a test application with a more complex search space. We will likely split up the work in the same way as for Milestone 1, as we will be familiar with our respective. Again, this task may prove too big to accomplish by Milestone 2, in which case it will carry over into Milestone 3.

Our goals for Milestone 3 will be largely dependant on our progress in previous Milestones 1 and 2. In any case, we hope to produce an interesting and non-trivial auto-tuning program. Depending on time, we may also do further analysis on this program, including comparisons of different search strategies and of results of our auto-tuner to related work such as ActiveHarmony.

## References


[1] Jeffrey K. Hollingsworth and Ananta Tiwari. *Performance Tuning of Scientific Applications*. CRC Press, University of Maryland, College Park, 2010.

[2] Benjamin Ylvisaker and Scott Hauck. Probabalistic auto-tuning for architectures with complex constraints. *EXADAPT '11*, pages 22–33, 2011.