

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

**Can Search Algorithms Save
Large-scale Automatic Performance Tuning?¹**

Prasanna Balaprakash, Stefan M. Wild, and Paul D. Hovland

Mathematics and Computer Science Division

Preprint ANL/MCS-P1823-0111

January 2011

¹Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439. This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

Contents

1	Introduction	1
2	The autotuning search problem	3
2.1	Tuning parameters	3
2.2	Performance objectives	4
2.3	Optimization challenges	4
3	Derivative-free optimization algorithms for autotuning	5
3.1	Random search	6
3.2	Genetic algorithms	7
3.3	Nelder-Mead simplex method with rounding	7
3.4	Modified Nelder-Mead algorithm	7
4	Illustrative experiments	8
4.1	Test problems	9
4.2	Experimental results	10
5	Conclusions and future directions	12

Can search algorithms save large-scale automatic performance tuning?¹

Prasanna Balaprakash, Stefan M. Wild, and Paul D. Hovland

Abstract

Empirical performance optimization of computer codes using autotuners has received significant attention in recent years. Given the increased complexity of computer architectures and scientific codes, evaluating all possible code variants is prohibitively expensive for all but the simplest kernels. One way for autotuners to overcome this hurdle is through use of a search algorithm that finds high-performing code variants while examining relatively few variants. In this paper we examine the search problem in autotuning from a mathematical optimization perspective. As an illustration of the power and limitations of this optimization, we conduct an experimental study of several optimization algorithms on a number of linear algebra kernel codes. We find that the algorithms considered obtain performance gains similar to the optimal ones found by complete enumeration or by large random searches but in a tiny fraction of the computation time.

1 Introduction

The pervasiveness of scientific computing in all applications has contributed to rises in the complexity and volume of codes run on high-performance computing architectures. At the same time, features such as heterogeneous cores, memory hierarchy, memory access times, multithreading, and processor-specific instructions have made these architectures increasingly complex. As a consequence, customary compiler optimization techniques are unable to achieve the levels of performance gains they once obtained. A promising approach to overcome the shortcomings of compiler optimization is *autotuning*. This approach consists of identifying relevant code optimization techniques (such as loop unrolling, register tiling, loop vectorization), assigning a range of parameter values using hardware expertise and application-specific knowledge, and then either enumerating or searching this parameter space to find the best-performing parameter configuration for the given architecture.

This tuning is no longer a one-time procedure: as codes and architectures evolve, scientific applications are moved from one architecture to another, and the empirical performance often suffers unless the code is retuned. Since the number of code variants grows exponentially with the number of tuning parameters, complete enumeration becomes prohibitively expensive. Thus algorithms capable of selecting high-performing parameter configurations

¹Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439. This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

in small computational budgets are vital to the effectiveness of large-scale performance tuning.

Most autotuning approaches described in the literature run empirical performance evaluations on a target machine to find good parameter values for that machine.

To find good parameter configurations, some autotuners perform complete enumeration either of all possible parameter configurations or of a pruned set of parameter configurations obtained by exploiting expert knowledge and architecture-specific and/or application-specific information. Examples include application-specific autotuners such as lattice Boltzmann computations [1], stencil computations [2], and matrix multiplication kernels [3, 4]. The main drawback of these autotuners is scalability: as codes and architectures become more complex, the number of tunable parameters and parameter configurations grows rapidly unless aggressive pruning is done.

Other autotuners address the issue of scalability by using search methods to select a parameter configuration. In this paper, we refer to *search* as any process short of complete enumeration. We will also call this *optimization*; hence, in the remainder of the paper, the term optimization will refer to mathematical optimization and should not to be confused with the specific problem of compiler optimization. Several optimization algorithms have been investigated for autotuning. ATLAS [5] uses an orthogonal search algorithm to provide high-performance codes for BLAS. Fursin et al. [6] adopted sequential and random search. Kisuki et al. [7] investigated the effectiveness of random search, pyramid search, window search, a genetic algorithm, and simulated annealing. Seymour et al. [8] analyzed the effectiveness of random search, Nelder-Mead simplex method, a genetic algorithm, simulated annealing, particle swarm optimization, and orthogonal search for empirical performance optimization. In these three works, the authors found that simple random search was more effective than most of the other algorithms considered. Qasem et al. [9] deployed a pattern search and showed that it was more effective than the random search when the search space is large. Norris et al. developed Orio [10, 11], a general-purpose, annotation-based autotuner that can deploy a simplex method, simulated annealing, and a genetic algorithm. Tiwari et al. [12] designed a simplex search technique inspired by a parallel rank-ordering method [13] in their autotuning framework.

In each case, the adoption of optimization is shown to obtain significant performance gains over default configurations of the tested codes. In this paper we provide perspectives on the autotuning search problem and the potential of derivative-free optimization algorithms. In Section 2 we formulate the autotuning search problem from a mathematical optimization standpoint and describe some of the challenges characteristic to this problem. By providing a precise definition of this optimization problem, one can more readily employ the many advanced optimization algorithms that have been developed. A small sample of optimization algorithms for solving tuning problems for which enumeration is infeasible is summarized in Section 3. Optimization algorithms must be carefully adapted to the autotuning problem domain. We provide evidence of this in Section 4, illustrating the differences in effectiveness between a naïve adaptation of the Nelder-Mead simplex algorithm and a more careful adaptation to the autotuning domain. Our experimental study suggests

that modern derivative-free optimization algorithms, which are even more powerful than the ones considered here, could efficiently deliver high-performing code variants.

2 The autotuning search problem

We now establish notation, formally define the class of performance tuning problems we consider, and attempt to bridge terminologies used by the mathematical optimization and performance-tuning communities. A performance-tuning problem can be modeled in many ways; problem formulation is a critical component, which should not be underestimated in any automated tuning process. Below we assume that we are given a well-defined performance-tuning problem, defined by a single performance metric and a set of feasible tuning parameters, and pose this as the mathematical optimization problem

$$\min_x \{f(x) : x = (x_{\mathcal{I}}, x_{\mathcal{B}}, x_{\mathcal{C}}) \in \mathcal{D} \subset \mathbb{R}^n\}. \quad (1)$$

2.1 Tuning parameters

We denote the n tuning parameters by a vector of *decision variables* $x = (x_{\mathcal{I}}, x_{\mathcal{B}}, x_{\mathcal{C}}) \in \mathbb{R}^n$, where we have explicitly partitioned x with the (possibly empty) index sets \mathcal{I} , \mathcal{B} , and \mathcal{C} , respectively denoting integer, binary, and continuous variables:

- $x_{\mathcal{I}}$ The variables $x_{\mathcal{I}}$ take integer values and can model any discrete parameters with a natural ordering. Examples include loop unroll factors, loop blocking sizes, and cache sizes (e.g., using 2^{x_i} for $x_i \in \{0, \dots, 10\}$).
- $x_{\mathcal{B}}$ The binary variables $x_{\mathcal{B}}$ take values in $\{0, 1\}$ and can model true/false or on/off decisions, for example, compiler flags, multicore parallelization, and scalar replacement transformations. Binary variables can be viewed as a subset of integer variables, but we reserve a separate term for them here as a reminder that one may want to explicitly handle parameters whose values have no special ordering.
- $x_{\mathcal{C}}$ Continuous variables $x_{\mathcal{C}}$ include parameters chosen from a continuum, though their realization and implementation in the tuning process may practically limit them to a discrete set of floating point numbers. Such parameters are typically found within the code being tuned, for example, a tolerance for an internal iterative solver.

It is the task of an autotuner tool or user to determine both the set of decision variables (tuning parameters) and an allowable range of values for these variables. In optimization, the set of allowable parameter configurations, \mathcal{D} , is called the *feasible set*. In practice, each of the tuning parameters is bounded, and the parameter values corresponding to a default code implementation are allowable, so that the set \mathcal{D} is bounded and nonempty. When there are no continuous variables, $\mathcal{C} = \emptyset$, \mathcal{D} denotes a discrete feasible set consisting of $|\mathcal{D}| < \infty$ distinct points.

The feasible set \mathcal{D} is defined by a set of constraints on the decision variables x that are typically independent of a run of the associated code variant, such as bound (e.g.,

$1 \leq x_i \leq 10$), linear (e.g., $\sum_{b \in \mathcal{B}} x_b \leq 256$), and nonlinear (e.g., $\sum_{i \in \mathcal{I}} x_i^2 = 25$) constraints. For these constraints the time required to verify the feasibility ($x \in? \mathcal{D}$) of an arbitrary point $x \in \mathbb{R}^n$ is negligible relative to the time required to evaluate the objective $f(x)$. More general constraints could, however, require execution of the code (e.g., $\{x : [\text{number of cache misses given } x] \leq 100\}$) and could be as expensive to evaluate as the objective.

We note that categorical variables, such as a compiler type or loop order, can be modeled through a combination of binary variables and constraints, for example, $\{x_b \in \{0, 1\}^3 : \sum_{b=1}^3 x_b = 1\}$ could denote that one of three compiler types must be selected.

2.2 Performance objectives

The objective f typically represents some empirical performance metric for the given code. This could include the execution time on a test input (or distribution of inputs), the compile time, or some combination of metrics. We assume that there is a single objective, though multiple simultaneous objectives could be considered in a more general setting and fall under the umbrella of *multiobjective optimization*. The objective f is also implicitly defined by the architecture on which the code is executed. Objectives \tilde{f} to be maximized can be analyzed by considering $f = -\tilde{f}$.

Ideally, the domain \mathcal{D} has been constructed so that the objective f can be evaluated at any feasible $x \in \mathcal{D}$, but for some applications this may not be known a priori. For example, one could encounter an x for which the resulting code does not compile or experiences some failure (e.g., a segmentation fault or producing an incorrect output) when executed. The set of such failures is called a *hidden constraint* by the optimization community. One way of handling this kind of infeasibility is with an *extreme barrier* approach [14], whereby we assign the objective value ∞ to any such infeasible point. Clearly Eq. (1) is well-posed only when there is some feasible point that does not result in a failure; that is, $\{x \in \mathcal{D} : f(x) < \infty\}$ must be nonempty. Typically this feasible point is the default code variant.

We say that an objective f is deterministic if for a fixed input x it always returns the same value $f(x)$. Even if the operations in a generated code are deterministic, the state of the machine may result in variability (measurement error) in the $f(x)$ obtained for the performance metrics typically of interest. As a result, performance metrics are often nondeterministic, and f in Eq. (1) should formally be the metric’s expectation or a similar deterministic measure. Optimization approaches usually either directly work with nondeterministic objectives or treat the objective as deterministic by assigning $f(x)$ the first value returned or a finite sample mean (median, etc.) at a given x .

2.3 Optimization challenges

Even if the performance objective f is assumed deterministic, Eq. (1) is a nonconvex *mixed integer nonlinear program* (MINLP), one of the most challenging problems in optimization. Furthermore, when the discrete variables ($x_{\mathcal{B}}, x_{\mathcal{I}}$) are held constant, one cannot expect that f is continuous in $x_{\mathcal{C}}$; for example, changing a tolerance for a Krylov solver may result in jumps in its empirical run time associated with the discrete number of iterations needed

to satisfy this tolerance. It is also impossible to evaluate f when the discrete variables are *relaxed* to be continuous (e.g., when $x_i \in \{0, \dots, 3\}$ is relaxed to $0 \leq x_i \leq 3$). For example, an unroll factor of 2.3 is meaningless.

Mathematical optimization typically relies on and benefits from algebraic expressions for derivatives of relaxations of f with respect to the decision variables, $\nabla_x f$. However, *derivative-free optimization methods* [15] are a special class of methods that seek solutions when such derivatives are unavailable to the method. These methods are commonly used for solving blackbox optimization problems because they rely only on function evaluations.

A feature of autotuning problems that an optimization algorithm may seek to exploit is that the time required to evaluate f is often not the same for different values of $x \in \mathcal{D}$. However, often this information is not known prior to the evaluation; for example, if f represents the run time of a code and compile time is negligible, the objective evaluation is quickest precisely at solutions of Eq. (1). On the other hand, evaluation of f at x may also require that a new code be generated and compiled, and the time to do the underlying transformations and compilation could potentially be well approximated by simple functions of the decision variables x . These issues require further research and will be increasingly important when approximate solutions to Eq. (1) are demanded within a short computation time.

An additional challenge is that there could be many distinct basins to which an optimization method could be attracted. We say that a point $x \in \mathcal{D}$ is *locally optimal* with respect to a neighborhood $\mathcal{N}(x)$ if $f(x) \leq f(y)$ for all $y \in \mathcal{D} \cap \mathcal{N}(x)$. Multimodality of the objective f and discontinuities in the search space due to constraints and discrete variables can contribute to there being multiple locally optimal points (for any reasonable definition of \mathcal{N}) that do not solve Eq. (1). Finding a global solution to Eq. (1) (i.e., when the neighborhood contains all of \mathcal{D}) is thus an immensely challenging problem, typically solvable only by complete enumeration unless further assumptions are made.

3 Derivative-free optimization algorithms for autotuning

Despite the challenges detailed in the previous section, there is evidence that optimization algorithms can efficiently find approximate solutions to Eq. (1). For the case of solely continuous variables ($|\mathcal{C}| = n$), the studies in [14, 16, 17] found optimization effective at finding good tuning parameters, even in very large feasible domains. Similarly, methods other than random search proved effective on the smaller, discrete domain problems considered in [9, 12, 13]. The goal of any optimization algorithm is to find parameter values that perform well (relative to the best performance attainable within \mathcal{D}) in relatively little time.

A fundamental limitation of many optimization algorithms considered in the literature is that they are likely to fail if there is absolutely no correlation between neighboring parameter values and their objectives. Despite potentially sharp discontinuities and disconnected feasible regions, our experience suggests that there will almost always be some structure in the objectives of interest. For example, Figure 1 illustrates the mean run time on a matrix-matrix-multiplication kernel with three variables taking integer variables in $\{1, \dots, 10\}$.

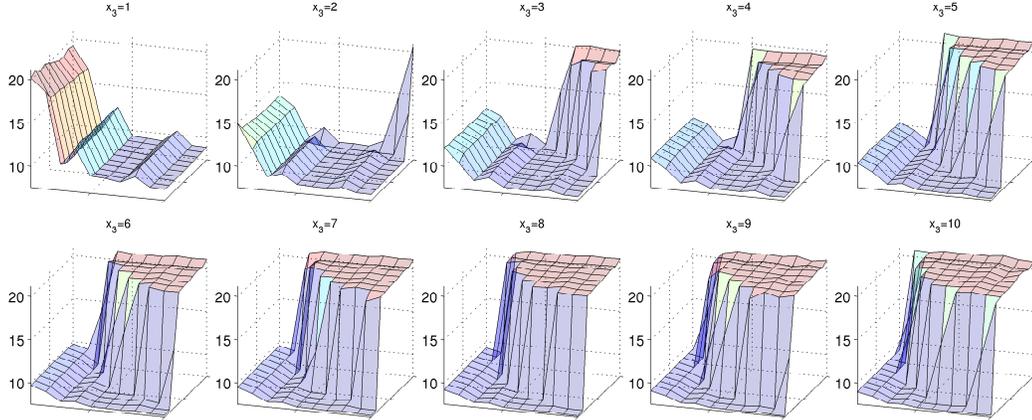


Figure 1: Surface plots of the mean run time of a matrix multiplication kernel as a function of three loop unrolling parameters.

The objective appears to be piecewise smooth; and, despite a pronounced discontinuity at $x_1 + x_2 + x_3 \approx 22$ and ridges in each 2-d slice, most descent-based approaches in the relaxed space would find an objective value close to the optimal one. Given behavior such as that illustrated in Figure 1, one is particularly tempted to try optimization algorithms, traditionally applied in continuous domains, that exploit some degree of smoothness on autotuning problems. When this smoothness is present, these methods typically require fewer evaluations than do more general algorithms.

We now briefly describe the four methods used in the next section on problems with only discrete variables. Each of these methods would benefit from adapting its internal parameters for each particular problem, and the amount of time one should devote to this tuning relative to the actual search is an important consideration for any automatic tuning package. For simplicity, we have fixed default settings for the optimization algorithms throughout, so that no overhead is incurred in tuning the optimization algorithms. In addition to criteria listed below, each algorithm can terminate when a finite budget of evaluations (or limit on the wall time) is reached.

3.1 Random search

We consider random search (RS) over the feasible domain \mathcal{D} without replacement. For discrete sets \mathcal{D} , this means that on iteration k , each $x \in \mathcal{D}$ not already selected has probability $\frac{1}{|\mathcal{D}|-k+1}$ of being selected as the point $x^{(k)}$. Having completed enumeration, the algorithm terminates after $|\mathcal{D}|$ iterations with the global minimum.

3.2 Genetic algorithms

Genetic algorithms (GAs) remain immensely popular heuristics for challenging global search.¹ GAs iteratively modify a population of solutions by applying a series of evolutionary operations such as reproduction, recombination, and mutation. As an illustration of a popular global search technique, we use a simple GA based on [18, 19]. This algorithm was originally proposed for continuous optimization with boundary constraints and is adapted to our setting by modifying, through rounding, the mutation operator to generate only integer points.

3.3 Nelder-Mead simplex method with rounding

A number of researchers have investigated the effectiveness of the Nelder-Mead (NM) simplex method for autotuning (see e.g., [8, 10, 11]). To solve unconstrained continuous optimization problems with n tuning parameters, the algorithm works with a simplex of $n + 1$ vertices. At each iteration, the simplex moves away from less promising regions of the search space using reflection, expansion, contraction, or shrink operators. The implementation we use in this paper is based on [20]. In order to handle the integer parameters, the decision point is rounded to the nearest integer only during the objective function evaluation. Moreover, when a boundary constraint for a parameter is violated, the cost function is simply a penalty related to the constraint violation; no expensive evaluation is charged to the algorithm. We use this algorithm to illustrate a naïve application of a continuous algorithm on discrete problems. This method is similar to the Nelder-Mead simplex method with rounding used by Seymour et al. [8].

3.4 Modified Nelder-Mead algorithm

To handle bounded integer parameters in a more direct way, we applied the following modifications to the standard Nelder-Mead method. First, when a value for a parameter is outside of its bounds, we perform a projection to the bound instead of penalizing the objective. Second, we force the simplex to move only on integer values by rounded versions of the reflection, expansion, contraction, and shrink operations. As a result of these modifications, the contraction and shrink operations may leave a vertex unchanged. In this case, the vertex is replaced by an unvisited neighbor of the best (lowest function value) vertex. We use the unit neighborhood $\mathcal{N}(x) = \{y : \|x - y\|_\infty \leq 1\}$ and note that there are $3^n - 1$ neighbors (excluding x); in two dimensions, the neighbors of (3,3) are $\{(3, 2), (3, 4), (2, 3), (4, 3), (2, 2), (4, 4), (2, 4), (4, 2)\}$. The modified Nelder-Mead method (mNM) thus can escape from a point that is not locally optimal with respect to \mathcal{N} . Because it performed well in the tests of the next section, we describe this method below. The mNM method differs from NM only through the modifications highlighted in boldface.

¹A Google Scholar search for *genetic algorithm* yields nearly as many results as one for *optimization algorithm*.

- Set the parameters α , β , and γ ; compute the objective function values on an initial simplex of $n + 1$ vertices.
- Perform the following steps for each iteration k until termination criteria are met:
 - Step 1, Centroid:** Label the $n + 1$ vertices so that $f(v_0^k) \leq f(v_2^k) \leq \dots \leq f(v_n^k)$
 - Set $\bar{v}^k = \frac{1}{n} \sum_{i=1}^n v_i^k$.
 - Step 2, Reflection:** Set $v_r^k = (1 + \alpha)\bar{v}^k - \alpha * v_n^k$
 - $v_r^k = \mathbf{round}(v_r^k)$
 - If $f(v_0^k) \leq f(v_r^k) < f(v_{n-1}^k)$, replace v_n^k with v_r^k , and return to step 1
 - Else if $f(v_r^k) < f(v_0^k)$, go to step 3
 - Else go to step 4.
 - Step 3, Expansion:** Set $v_e^k = \gamma v_r^k + (1 - \gamma)\bar{v}^k$
 - $v_e^k = \mathbf{round}(v_e^k)$
 - If $f(v_e^k) \leq f(v_r^k)$, replace v_n^k with v_e^k , and return to step 1.
 - Step 4, Contraction:** Set $v_t^k = v_n^k$
 - If $f(v_r^k) \leq f(v_t^k)$, set $v_t^k = v_t^k$
 - $v_c^k = \beta v_t^k + (1 - \beta)\bar{v}^k$
 - $v_c^k = \mathbf{round}(v_c^k)$
 - **If $v_t^k = v_c^k$, set v_c^k to an unvisited neighbor of v_0^k**
 - If $f(v_c^k) \leq f(v_n^k)$, replace v_n^k with v_t^k , and return to step 1.
 - Step 5, Shrinking:** For $i = 1, \dots, n$, do:
 - * $vs_i^k = (v_0^k + v_i^k)/2$
 - * $vs_i^k = \mathbf{round}(vs_i^k)$
 - * **If $vs_i^k = v_i^k$, set vs_i^k to an unvisited neighbor of v_0^k**
 - * Set $v_i^k = vs_i^k$, and return to step 1.

4 Illustrative experiments

For code generation and transformation, in our experiments we use Orio [10, 11], an extensible annotation-based performance-tuning tool. Using annotated C code as input, Orio automatically generates multiple transformed versions of the given code with respect to the tunable parameter values specified in the annotation. Orio currently supports a number of performance optimization techniques, including simple loop unrolling, memory alignment optimization, loop unroll/jamming, loop tiling, loop permutation, scalar replacement, register tiling, loop-bound replacement, array copy optimization, multicore parallelization (using OpenMP), and other architecture-specific optimizations (e.g., generating calls to SIMD intrinsics on Intel and Blue Gene/P architectures). We use Orio only as a source transformation tool: given values for the tuning parameters, Orio is used to transform the source code and evaluate the resulting empirical performance. However, Orio also makes available standard implementations of some of the algorithms considered here if search in the parameter space is desired.

Table 1: Collection of problems from the Orio test suite.

Kernel	Operation	Transformations			$ \mathcal{D} $	Time (s)		
		n_i		n_b		$\hat{\mu}_{init}$	$\hat{\sigma}_{init}$	
ATAX	matrix transpose & vector multiplication	4	UJ	4	SR,LP,LV	1e+07	.181	2.46e-4
BiCG	sub-kernel of BiCGStab linear solver	3	UJ	5	SR,LP,LV	9e+05	.149	1.04e-4
DGEMV	scalar, vector, & matrix multiplication	15	UJ,RT	5	SR,LP,LV	5e+23	.011	4.37e-5
FDTD4d2d	finite-difference time-domain kernel	7	UJ	2	SR,LV	9e+10	.123	2.34e-4
GEMVER	vector multiplication & matrix addition	5	UJ	7	SR,LP,LV	3e+09	.370	1.61e-4
GESUMMV	scalar, vector, & matrix multiplication	2	UJ	3	SR,LP,LV	7e+03	.158	4.73e-5
LU	matrix factorization	4	UJ	3	SR,LV	1e+07	.034	4.58e-6
MM	matrix multiplication	3	UJ	0		3e+04	.351	4.89e-5
Tensor	tensor matrix multiplication	5	UJ	0		2e+04	.130	2.55e-5

4.1 Test problems

Here we study the nine linear algebra kernel codes shown in Table 1, each implemented in C and considered with the test input, set of tuning parameters, and feasible domain given by the Orio test suite. Other tuning parameters could be considered for each kernel; we use those from the Orio test suite solely for illustrative purposes and note that a few of these kernels have previously been studied for autotuning in [21]. For each problem we report the number of tunable integer (n_i) and binary (n_b) parameters and the underlying transformations; the acronyms UJ, SR, LP, LV, and RT denote loop unrolling, scalar replacement, loop parallelization, loop vectorization, and register tiling transformations, respectively. Each integer parameter takes a value between 1 and 30; with the exception of **Tensor**, all problems are otherwise unconstrained. For **Tensor** there are nonlinear constraints that are trivial to evaluate (e.g., $x_1x_2x_3x_4 + x_1x_2x_3x_5 + x_4x_5 \leq 130$) and for which we use an extreme barrier approach as described in Section 2.

Experiments are carried out on dedicated nodes of Fusion, a 320-node cluster at Argonne National Laboratory, comprising 2.6 GHz Pentium Xeon processors with 36 GB of RAM, under the stock Linux kernel version 2.6.18 provided by RedHat.

Each objective evaluation entails running the generated code 35 times, with the function value $f(x)$ with parameter configuration x given by the average computation time over these 35 runs. This averaging is done in part to obtain uniform system conditions and hence reduce nondeterministic variations in the objective. The columns $\hat{\mu}_{init}$ and $\hat{\sigma}_{init}$ in Table 1 denote the estimated mean and standard deviation of the run time for the 35 runs used in the evaluation of f at the initial parameter configuration. A measure of the relative noise, $\hat{\sigma}_{init}/\sqrt{35}\hat{\mu}_{init}$, shows that each f is stable to 3 or 4 significant digits. For all of these kernels, we found that the median of the 35 runs was close to the mean but systematically lower, most likely due to effects such as cold caches. Here we take the risk neutral approach of using the mean, but other objectives (such as the median or other quantile-based measures) could be chosen based on the ultimate goals of the performance-tuning process.

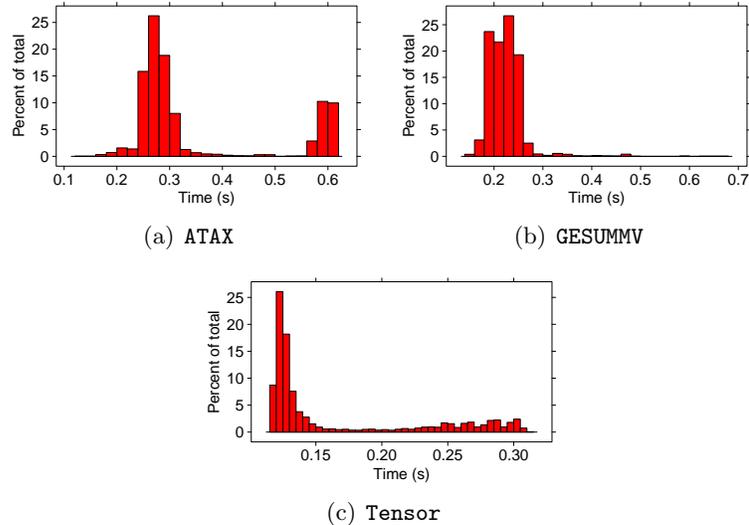


Figure 2: Histograms of objective values from 5,000 random code variants in \mathcal{D} .

Also of interest is the variation in f (and hence the run times) throughout the feasible domain \mathcal{D} . In Table 2, we report the mean (RS) and standard deviation ($\hat{\sigma}_{all}$) of the objective values f for 1,000 random feasible parameter configurations when the evaluation was successful (i.e., a hidden constraint was not violated). Clearly, larger speedups are possible only for kernels with large variations $\hat{\sigma}_{all}$ throughout the domain; the time required for optimization may not be productive for some applications where $\hat{\sigma}_{all}$ is very small. Though these variations of milliseconds may seem small, they quickly add up when these kernels are called millions of times in a large scientific application code.

Figure 2 shows a histogram of the objective values obtained on 5,000 random parameter configurations for **ATAX**, **GESUMMV**, and **Tensor**. We see that for **ATAX** and **GESUMMV** the number of high-performing parameter configurations is low compared to that for **Tensor**. Based on the histograms and $\hat{\sigma}_{all}$ values, we hypothesize that other optimization algorithms will be more effective than RS on kernels such as **ATAX** and **GESUMMV**, where $\hat{\sigma}_{all}$ is large and the percentage of high-performing parameter configurations is low. On the other hand, we expect that RS will perform well for kernels such as **Tensor**, where there are many high-performing parameter configurations, and **DGEMV**, where $\hat{\sigma}_{all}$ is small.

4.2 Experimental results

The optimization methods described in Section 3 are implemented and run in MATLAB Version 7.9.0.529 (R2009b). We adopt default parameter values for the Nelder-Mead simplex methods and the genetic algorithm. All algorithms start with the initial configuration obtained by setting each variable to its lower bound and do not differentiate between binary and integer variables. For the simplex methods, the size of the initial simplex can have a significant impact on the methods' performance. For this study we set the initial simplex to be the coordinate-based simplex consisting of the initial point plus the n vertices obtained

Table 2: Average time per successful evaluation (in seconds) and number of failures due to hidden constraints.

Kernel	$\hat{\sigma}_{all}$	Average Time (Failures per 1000)		
		RS	GA	mNM
ATAX	1.3810e-01	0.3512 (0)	0.1643 (0)	0.1499 (0)
BICG	2.0650e-02	0.2001 (258)	0.1177 (21)	0.1138 (39)
DGEMV	1.5068e-03	0.0113 (469)	0.0073 (14)	0.0070 (77)
FDTD4d2d	3.2564e-02	0.0810 (941)	0.0456 (68)	0.0502 (69)
GEMVER	6.2733e-02	0.5364 (0)	0.4341 (0)	0.4305 (0)
GESUMMV	3.8130e-02	0.2207 (28)	0.1750 (0)	0.1796 (0)
LU	1.5300e-02	0.0914 (619)	0.0532 (43)	0.0367 (50)
MM	4.8550e-02	0.2691 (6)	0.1861 (0)	0.1741 (0)
Tensor	6.283e-02	0.1655 (0)	0.1273 (0)	0.1290 (0)

when each parameter is set to its upper bound and the rest are held fixed.

From an optimization viewpoint, it is customary to measure reduction in the number of code variant evaluations, as shown in Figure 3. In the context of autotuning, however, this can be a conservative measure, as discussed in Section 2. A more targeted optimization algorithm may require less time to do 1,000 evaluations than will a more exploratory algorithm like random search. We quantify this effect in Table 2 by reporting the average run time (for successful evaluations) across 1,000 evaluations performed by RS, GA, and mNM. In Figure 4, we illustrate how this effect changes the behavior seen in Figure 3 by plotting the best function value obtained as measured in evaluation time for the three kernels for which there were no failures.

For example, on *ATAX*, the results show that mNM and GA obtain high-quality parameter configurations that are significantly better than that of RS. With respect to the objective value of the initial parameter configuration, mNM and GA obtain speedups of 1.55 and 1.53, respectively, whereas RS obtains a speedup of 1.22. The attractive feature of mNM is that it obtains high-quality parameter configurations within very short search times—mNM requires 3.5 orders of magnitude less search time than RS. Since GA is a population-based approach, it spends more time exploring the domain and tends to be slower than mNM. We observe a similar behavior on the majority of the other kernels.

For validation purposes, we performed a complete enumeration on *MM* to measure the difference in the objective value between the globally optimal parameter configuration and the configurations obtained by mNM and GA. The results showed that the observed differences are small (0.001 seconds).

On *Tensor*, where the number of high-quality parameter configurations is very high, RS obtains a high-quality parameter configuration in a short search time. The poor performance of mNM is due to the fact that it gets stuck at a local minimum of poor quality, possibly a result of the nonlinear constraints on this problem.

In all the experiments, we found that NM stagnated at a poor quality point, often not a local minimum. This result clearly shows that a straightforward adaptation (rounding, in this case) of a continuous algorithm is unlikely to be successful on these types of problems and further customizations are required. This could explain why the Nelder-Mead algorithm used by Seymour et al. performed worse than random search in [8].

5 Conclusions and future directions

This paper consists of three main contributions. First, we have formulated the search problem encountered in autotuning as a mathematical, derivative-free optimization problem. Second, we have illustrated the potential for optimization algorithms to find high-performing tuning parameters in a short computation time. Third, our empirical study illustrates both that extreme care must be taken in how optimization algorithms are modified to this domain and that the problem characteristics can significantly impact the effectiveness of optimization algorithms for the autotuning task.

Our experimental results are encouraging for large-scale tuning problems where it is prohibitively expensive to perform complete enumeration of all possible code variants. The simple optimization algorithms considered examined only tiny portions of the search space – the modified Nelder-Mead method found a good solution after examining fewer than 0.000000000000000001% of the `ATAX` variants. We do not provide a specific recommendation for an end-all optimization algorithm for autotuning because we expect that the results found here will improve as these algorithms are themselves tuned and adapted for this setting. Furthermore, benchmark studies such as [22] on piecewise smooth problems indicate that other local optimization algorithms could do even better than the Nelder-Mead method that performed well here. As we have shown, however, these standard local methods from continuous optimization will need to be carefully specialized for this setting to overcome the challenges of discrete variables, hidden constraints, and otherwise sharp discontinuities.

Further challenges persist, primarily concerning automatically developing optimization problem formulations that are particularly amendable to subsequent optimization algorithms. When a code doesn't require all of a cluster's resources, it is useful to evaluate several different code variants in parallel, a topic explored in [12, 13]. We also anticipate substantial benefit from methods that take into account the heterogeneity of times to evaluate different code variants and the availability of fast estimates of these times (e.g., an estimate of the compile time could be obtained based on loop unrolling parameter values).

Acknowledgments

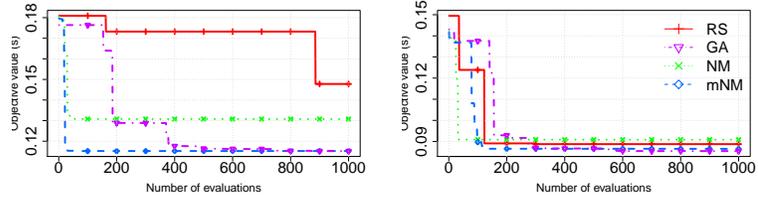
We are grateful to Boyana Norris for helpful discussions and to the Laboratory Computing Resource Center at Argonne National Laboratory.

References

- [1] S. Williams, J. Carter, L. Oliker, J. Shalf, K. Yelick, Optimization of a lattice Boltzmann computation on state-of-the-art multicore platforms, *Journal of Parallel and Distributed Computing* 69 (9) (2009) 762–777.
- [2] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, K. Yelick, Optimization and performance modeling of stencil computations on modern microprocessors, *SIAM Review* 51 (1) (2009) 129–159.
- [3] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, J. Demmel, Optimization of sparse matrix-vector multiplication on emerging multicore platforms, *Parallel Computing* 35 (3) (2009) 178–194.
- [4] J. Shin, M. W. Hall, J. Chame, C. Chen, P. D. Hovland, Autotuning and specialization: Speeding up matrix multiply for small matrices with compiler technology, in: *Proceedings of the Fourth International Workshop on Automatic Performance Tuning*, Japan, 2009.
- [5] R. C. Whaley, J. J. Dongarra, Automatically tuned linear algebra software, in: *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing, SC '98*, Washington, DC, 1998, pp. 1–27.
- [6] G. G. Fursin, M. F. P. O'Boyle, P. M. W. Knijnenburg, Evaluating iterative compilation, in: *Proceedings of the 15th Workshop on Languages and Compilers for Parallel Computing (LCPC'02)*, 2002, pp. 305–315.
- [7] T. Kisuki, P. M. W. Knijnenburg, M. F. P. O'Boyle, Combined selection of tile sizes and unroll factors using iterative compilation, in: *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, Washington, DC, 2000.
- [8] K. Seymour, H. You, J. Dongarra, A comparison of search heuristics for empirical code optimization, in: *Proceedings of the 2008 IEEE International Conference on Cluster Computing*, 2008, pp. 421–429.
- [9] A. Qasem, K. Kennedy, J. Mellor-Crummey, Automatic tuning of whole applications using direct search and a performance-based transformation system, *The Journal of Supercomputing* 36 (2) (2006) 183–196.
- [10] B. Norris, A. Hartono, W. Gropp, *Annotations for Productivity and Performance Portability*, Computational Science, Chapman & Hall CRC Press, Taylor and Francis Group, 2007, pp. 443–461.
- [11] A. Hartono, B. Norris, P. Sadayappan, Annotation-based empirical performance tuning using Orio, in: *Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium*, Italy, 2009.

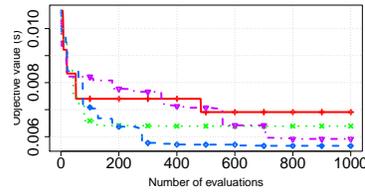
- [12] A. Tiwari, C. Chen, C. Jacqueline, M. Hall, J. K. Hollingsworth, A scalable auto-tuning framework for compiler optimization, in: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing, Washington, DC, 2009, pp. 1–12.
- [13] V. Tabatabaee, A. Tiwari, J. K. Hollingsworth, Parallel parameter tuning for applications with performance variability, in: Proceedings of the 2005 ACM/IEEE conference on Supercomputing, SC '05, Washington, DC, 2005.
- [14] C. Audet, D. Orban, Finding optimal algorithmic parameters using derivative-free optimization, *SIAM Journal on Optimization* 17 (3) (2006) 642–664.
- [15] A. R. Conn, K. Scheinberg, L. N. Vicente, Introduction to Derivative-Free Optimization, MPS/SIAM Series on Optimization, Society for Industrial and Applied Mathematics, Philadelphia, PA, 2009.
- [16] C. Audet, D. C.-K, D. Orban, Algorithmic parameter optimization of the DFO method with the OPAL framework, in: K. Naono, K. Teranishi, J. Cavazos, R. Suda (Eds.), *Software Automatic Tuning: From Concepts to State-of-the-Art Results*, Springer, 2010, pp. 255–274.
- [17] F. Hutter, H. H. Hoos, K. Leyton-Brown, T. Stützle, ParamILS: An automatic algorithm configuration framework, *Journal of Artificial Intelligence Research* 36 (2009) 267–306.
- [18] R. E. Dorsey, W. J. Mayer, Genetic algorithms for estimation problems with multiple optima, nondifferentiability, and other irregular features, *Journal of Business & Economic Statistics* 13 (1) (1995) 53–66.
- [19] M. Gordy, GA.m: A Matlab routine for function maximization using a genetic algorithm (1996).
- [20] N. J. Higham, Optimization by direct search in matrix computations, *SIAM Journal on Matrix Analysis and Applications* 14 (2) (1993) 317–333.
- [21] B. Norris, A. Hartono, E. Jessup, J. Siek, Generating empirically optimized composed matrix kernels from Matlab prototypes, in: G. Allen, J. Nabrzyski, E. Seidel, G. van Albada, J. Dongarra, P. Sloot (Eds.), *International Conference on Computational Science*, 2009, pp. 248–258.
- [22] J. J. Moré, S. M. Wild, Benchmarking derivative-free optimization algorithms, *SIAM Journal on Optimization* 20 (1) (2009) 172–191.

The submitted manuscript has been created by the University of Chicago as Operator of Argonne National Laboratory (“Argonne”) under Contract DE-AC02-06CH11357 with the U.S. Department of Energy. The U.S. Government retains for itself, and others acting on its behalf, a paid-up, nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.

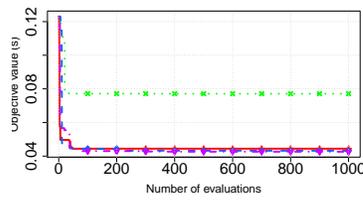


(a) ATAX

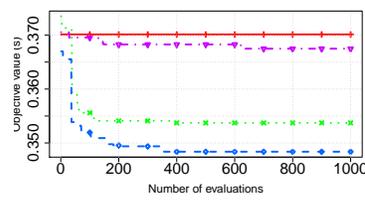
(b) BICG



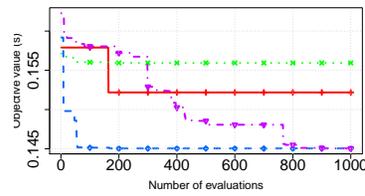
(c) DGEMV



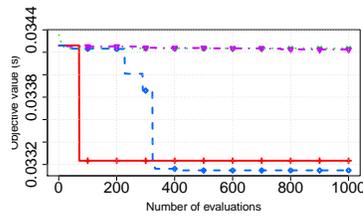
(d) FDTD4d2d



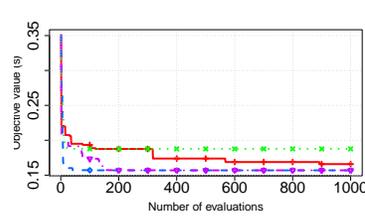
(e) GEMVER



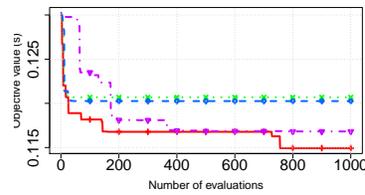
(f) GESUMMV



(g) LU



(h) MM



(i) Tensor

Figure 3: Best objective value obtained by each algorithm as a function of the number of (successful and unsuccessful) evaluations.

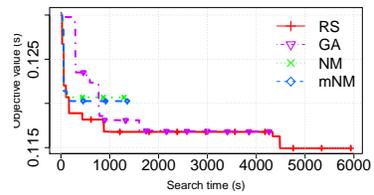
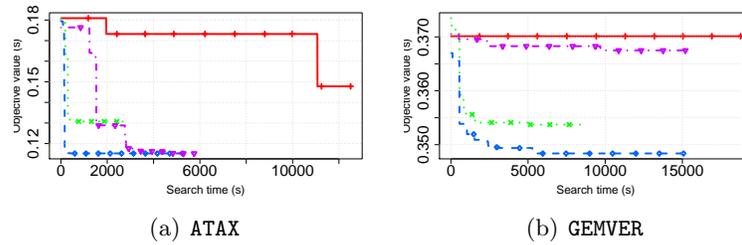


Figure 4: Best objective value obtained by each algorithm as a function of search time (markers every 100 evaluations) on kernels with no failures.