

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

**SPAPT:
Search Problems in Automatic Performance Tuning** ¹

Prasanna Balaprakash, Stefan M. Wild, and Boyana Norris

Mathematics and Computer Science Division

Preprint ANL/MCS-P1956-0911

January 2012

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	Related Work	3
3	Test Suite	4
4	Illustrative Experiments	7
4.1	Effect of cache misses and the impact of performance metric choice	7
4.2	Impact of the target machine	10
4.3	Performance objective density	10
4.4	Impact of input size	12
5	Conclusions and Future Directions	12

SPAPT: Search Problems in Automatic Performance Tuning ¹

Prasanna Balaprakash, Stefan M. Wild, and Boyana Norris

Abstract

Automatic performance tuning of computationally intensive kernels in scientific applications is a promising approach to achieving good performance on different machines while preserving the kernel implementation’s readability and portability. A major bottleneck in automatic performance tuning is the computation time required to test a large number of possible code variants, which grows exponentially with the number of tuning parameters. Consequently, the design, development, and analysis of effective search techniques capable of quickly finding high-performing parameter configurations have gained significant attention in recent years. An important element needed for this research is a collection of test problems that allow performance engineering and mathematical optimization researchers to conduct rigorous algorithmic development and experimental studies. In this paper, we describe a set of extensible and portable search problems in automatic performance tuning (SPAPT) whose goal is to aid in the development and improvement of search strategies. SPAPT contains representative serial code implementations from a number of lower-level performance-tuning tasks in scientific applications. We present an illustrative experimental study on several problems from the test suite. We discuss important issues such as modeling, search space characteristics, and performance objectives.

1 Introduction

The landscape of scientific application programming is undergoing rapid changes as a result of increasingly complex machines and the quest for high performance on these machines. Chasing performance gains through manual tuning becomes a complex and time-consuming process that is neither scalable nor portable. Automatic performance tuning (in short, *autotuning*), or empirical performance tuning, is a promising and viable approach to address the limitations of manual tuning. Autotuning involves three major phases: identifying code optimization techniques that are relevant to the given code and machine, assigning a range of parameter values using hardware expertise and application-specific knowledge, and searching the parameter space to find the best-performing parameter configuration for the given machine. In recent years, this has emerged as an effective approach to tune scientific kernels for both serial and multicore processors [1].

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.

A major bottleneck in large-scale autotuning is the prohibitively large computation time required when searching for high-performing parameter configurations in a large search space. Hence, popular search algorithms such as random search, Nelder-Mead, simulated annealing, and genetic algorithms are used to examine a small subset of possible configurations. In [2], we showed that the search problem arising in autotuning can be formulated as a mathematical optimization problem, and we illustrated the potential for mathematical optimization algorithms to find high-performing tuning parameters in a short computation time.

The primary obstacle for the mathematical optimization community to contribute algorithms for performance tuning is the high startup cost associated with developing mathematical formulations of autotuning problems and subsequently transforming, compiling, and running the corresponding codes. An easy-to-use test suite of well-defined mathematical optimization problems in autotuning addresses this issue. A mathematical optimization researcher can use the test suite to develop and test new numerical optimization algorithms. The results and insights obtained can provide recommendations for optimization algorithms based on particular problem characteristics. Given an autotuning task, a performance-tuning researcher can then adopt an optimization algorithm based on the search problem characteristics. Currently, doing so is difficult because few systematic tests of optimization algorithms exist for typical autotuning problems. In fact, recent successes of performance tuning in mathematical optimization have focused on obtaining parameters for other optimization algorithms (e.g., [3]), these codes being most familiar to optimizers.

A rich history in mathematical optimization of sets of benchmark problems exists. Examples include the Moré-Garbow-Hillstom problems for unconstrained optimization [4]; the more general CUTER set [5] (a subset of which was used as the inputs in [3]); and the smooth, noisy, and nonsmooth problems in [6]. These benchmarks are attractive for several reasons, including (1) providing a rigorous definition of a set of easily obtained problems; (2) absolving algorithm developers from controversial decisions related to problem formulation, scaling, and input parameter decisions; (3) mitigating particularly unusual behavior (e.g., seen on only a single problem); and (4) defining a self-contained, fixed set to avoid criticisms of including only problems that show favorable aspects of a particular algorithm. In addition to these characteristics, an ideal set would be large enough to yield diverse problems (rather than containing a single problem) but not too large to be prohibitively expensive, which would prevent one from running the benchmark set in its entirety. As evidenced by their citation counts, these benchmark sets are used extensively by the optimization community. The usual benchmarking caveats apply: performance of an optimization algorithm on the set is not a guarantee that it will perform similarly on all other problems, and hence one should avoid both “overfitting” and making extrapolations far beyond the set. However, results on the benchmark sets can still provide valuable feedback to developers on the algorithmic features expected to be most important; and they are a first step in developing, for example, specialized algorithms for classes of performance-tuning problems.

In this paper, we present a collection of extensible and portable search problems in **automatic performance tuning** (SPAPT). It comprises representative problems from a num-

ber of lower-level, serial performance-tuning tasks in scientific applications. As a starting point, we fill the initial test suite with primarily dense linear algebra kernels because they are ubiquitous and well understood by the performance-tuning and scientific computing communities. We implement problems in a format that can be readily processed by Orío [7], a recently developed autotuning software framework. By making Orío a tool for transforming and running code for SPAPT and defining specific search problems, our first goal is to attract the mathematical optimization community to help advance the field of autotuning. With the benchmark set, our second goal is to enable performance engineering and mathematical optimization researchers to conduct rigorous algorithmic development and experimental studies on search algorithms in autotuning.

SPAPT comprises kernel codes that run on a single node. There are two main reasons for this design choice. First, we wanted SPAPT to be an easily usable and portable test suite from the perspective of the mathematical optimization community. As a first step, we did not include parallel codes that need large computational clusters and/or leadership-class machines because the mathematical optimization researchers might have limited access to these machines. However, given that modern single-node machines—including desktops and laptops—come with multiple cores, search problems in SPAPT contain OpenMP directives as code transformation techniques. Second, since clusters are built by connecting single nodes via a network, any improvements in single-node performance may help with overall performance of certain applications. Third, single-node performance tuning is relevant in a number of kernels where the communication cost between the processor and the memory hierarchy is a bottleneck for the performance.

The two major contributions of the paper are as follows: (1) SPAPT, a first attempt to bring the optimization and performance-tuning research communities together and enable interdisciplinary research; and (2) ready-to-use test suite of well-defined mathematical optimization problems, each consisting of a relatively well-known kernel, an input size, a set of tunable decision parameters, a feasible set of possible parameter values, and an initial configuration of these parameters and constraints. With SPAPT, we perceive that numerical optimization and autotuning research communities will get a new class of problem to tackle and effective optimization algorithms, respectively.

2 Related Work

Balaprakash et al. [2], Kisuki et al. [8], Qasem et al. [9], Seymour et al. [10], Shin et al. [11], and Tiwari et al. [12] used a number of linear algebra kernels for autotuning. Pouchet [13] adopted a collection of reference implementations, which comprises linear algebra kernels, solvers, stencils, and data-mining codes. These codes have pragma delimiters for OpenMP and loop bounds for autotuning with a polyhedral model. Norris et al. [7] used a collection of linear algebra kernels, solvers, and stencils. These are parameterized codes that were used to test the effectiveness of Orío. In all these works, the kernels are often parameterized to illustrate the effectiveness of autotuning, but there is limited empirical analysis of the search algorithms applied to kernels with a large number of parameters that have wide ranges of

input sizes. Recently, Kaiser et al. [14] proposed the TORCH testbed, a set of reference kernels to enable software and hardware codesign. These kernels are broadly classified into linear algebra, grid, spectral, particle, Monte Carlo, graphs, and sort kernels. The authors discuss possible code optimization strategies that can be applied to these kernels. Nevertheless, parameterization and search problem specifications are not part of the testbed. Kaiser et al. [14] argue that a number of existing test suites can be seen as reference implementations of one or more kernels from TORCH. Examples include EEMBC [15], HPC Challenge [16], Parboil [17], SPEC [18], NAS Parallel test suites [19], PARSEC [20], Rodinia [21], LINPACK [22], STREAM [23], STAMP [24], SPLASH [25], and pChase [26]. Although in principle these test suites can be parameterized and used for autotuning, none is developed specifically for evaluating the performance of the search problem in autotuning. Hence, a noticeable void exists in the literature of test suites of well-formulated search problems in autotuning.

The SPAPT set that we propose in this paper is based on [7, 13, 14] and comprises representative examples from dense linear algebra computations. However, SPAPT differs from other test suites in the following way: it is the only test suite in the autotuning literature that is exclusively designed for developing and benchmarking optimization algorithms. In SPAPT, we make only the search problem a transparent entity—one can easily integrate an optimization algorithm to tackle the search problem without knowing the fine details related to the code transformation techniques, compiler specifics, and target machine.

3 Test Suite

We use the term *kernels* to refer to (deeply) nested loops that arise frequently in a number of scientific application codes. Because they contribute significantly to the overall execution time, tuning these kernels significantly improves overall application performance [27]. A range of transformations can be applied to better utilize the memory hierarchy and to help exploit shared-memory parallelism on multicore machines. The SPAPT benchmark that we propose in this paper comprises 18 such kernels. These kernels are grouped into four groups. **Elementary linear algebra kernels** involve a set of mathematical computations performed on scalars, vectors, and matrices. Because of the wide range of applications that adopt these kernels, autotuning these kernels is a popular topic of research and development. In this group we have ten kernels that consist of elementary linear algebra operations such as vector/matrix/tensor multiplications and transposes; see Table 1 for a summary of the operations involved. **Linear solver kernels** find solutions to a system of linear equations. In this group, we have kernels from the BiCGStab linear solver (`BiCG`) and LU, which decomposes a matrix into a product of lower and upper triangular matrices. **Stencil code kernels** follow a regular pattern to access and update array elements. They are commonly used in implicitly and explicitly solving partial differential equations [28]. In this group, we have four kernels from ADI preconditioners (`ADI`), Jacobi 1-D (`Jacobi-1d`), Seidel stencil (`Seidel`), and 3-D stencils computations (`Stencil3d`). **Elementary statistical computing kernels** are here represented by correlation (`COR`) and covariance (`COV`) computations.

Table 1: Collection of test suite kernels.

Kernel	Operation	Parameters		$ \mathcal{D} $
		n_i	n_b	
Elementary linear algebra kernels				
ATAX	matrix transpose & vector multiplication	13	6	1.65e+14
DGEMV	scalar, vector & matrix multiplication	38	11	2.73e+30
FDTD4d2d	finite-difference time-domain kernel	25	5	7.06e+24
GEMVER	vector multiplication & matrix addition	18	6	7.26e+17
GESUMMV	scalar, vector, & matrix multiplication	8	3	1.56e+08
HMC	Hessian matrix computation kernel	7	8	1.01e+08
MM	matrix multiplication	10	4	1.83e+12
MVT	matrix vector product & transpose	6	6	1.38e+08
Tensor	tensor matrix multiplication	17	3	5.49e+16
TRMM	triangular matrix operations	20	5	5.33e+19
Linear solver kernels				
BiCG	subkernel of BiCGStab linear solver	9	4	9.33e+09
LU	LU decomposition	9	5	1.86e+10
Stencil code kernels				
ADI	matrix subtraction, multiplication, & division	16	4	6.05e+15
Jacobi-1d	1-D Jacobi computation	8	3	1.55e+08
Seidel	matrix factorization	12	3	6.86e+11
Stencil3d	3-D stencil computation	24	5	2.35e+23
Elementary statistical computing kernels				
COR	correlation computation	16	4	6.05e+15
COV	covariance computation	20	5	5.33e+19

They involve finding statistical relationships among a number of random variables, a task that is central to many statistical packages. The reference implementations are obtained from [13], where the author made a similar classification of kernels.

We take a search *problem* in SPAPT to mean a specific combination of a kernel, an input size, a set of tunable decision parameters, a feasible set of possible parameter values, and a default/initial configuration of these parameters for use by search algorithms. When combined with a specific machine and a single performance objective f , both discussed further in Section 4, this search problem is equivalent to the mathematical optimization problem

$$\begin{aligned}
 & \min_x f(x) \\
 & \quad x = (x_{\mathcal{B}}, x_{\mathcal{I}}) \in \Omega, \\
 & \text{such that } x_{\mathcal{B}_j} \in \{0, 1\}, \quad j = 1, \dots, n_b, \\
 & \quad \quad \quad x_{\mathcal{I}_j} \in \{l_j, \dots, u_j\}, \quad j = 1, \dots, n_i,
 \end{aligned} \tag{1}$$

where \mathcal{B} and \mathcal{I} denote a partitioning of the parameter vector x into n_b binary and n_i integer scalars, respectively. Details on modeling and formulating problems such as (1) are given in [2]. We denote the collective feasible set for a given problem by \mathcal{D} , which is

defined by three classes of constraints. **Bound constraints:** All the parameters of the search problems are constrained to lie within lower and upper bounds. Examples of these constraints include loop unroll jam, where the values are positive and take integer values up to an upper bound. **Known constraints:** We have two subclasses of known constraints. The first subclass is algebraic constraints, where 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 $f(x)$; for example, limiting two register tiling parameters RT_I, RT_J , to certain values satisfying $RT_I * RT_J \leq 150$. The second subclass is general constraints that require execution of the code and could be as expensive to evaluate as the objective; for example, power consumption of a code run < 90 W. In all these constraints a quantifiable measure of constraint violation is available. From a mathematical optimization perspective, this is an important measure since it can help the optimization algorithm move away from regions of infeasibility. **Hidden constraints:** These constraints are attributed to unsuccessful code evaluations that occur as a result of transformation, compilation, and runtime errors. While failure at the code transformation phase is relatively cheap, failure due to runtime errors is expensive. In all these cases, a nonbinary measure of violation is not available; hence, dealing with these constraints can be difficult.

From each tunable kernel, we generate four search problems. For example, for the ATAX kernel, we have ATAX.N.01, ATAX.N.02, ATAX.N.03, and ATAX.01.N.nb. The naming conventions take the following meaning: N.01 is the (reference) input size in ATAX.N.01; N.02 $>$ N.01 and N.03 $>$ N.01 are the input sizes in ATAX.N.02 and ATAX.N.03, respectively. Note that the reference input size is not limited to single-dimensional or square inputs; for nonsquare or multidimensional inputs, instead of N , we have $\{N_1, N_2, N_3, \dots\}$. ATAX.N.01.nb is obtained from ATAX.N.01 by fixing the value of all binary parameters to 0 (so that only integer decision parameters are considered; nb refers to “no binary parameters”). The reason for explicitly including nb problems is that they can serve as an entry point for continuous numerical optimization algorithms that treat integer parameters similar to real-valued ones.

We define the initial configuration of a problem as that obtained by setting each integer variable to its lower bound and each binary variable to 0 (false). Note that this corresponds to the base implementation without any code transformation and optimization. From a mathematical optimization standpoint, the starting point can be important. Other starting points can be used on these problems; but to provide a well-defined, platform-independent starting point, we set the lower bound as a default. In addition to the goals discussed in Section 1, these problems enable us to study the impact of input size on performance tuning and to analyze the smoothness in the search space (e.g., because binary decisions such as enabling or disabling OpenMP often create discontinuities in the search space).

Table 1 gives a high-level overview of each kernel. Whenever applicable, we adopt the following general-purpose, parameterized tuning directives: loop unroll/jamming (UJ), cache tiling (CT), register tiling (RT), scalar replacement (SR), array copy optimization (AC), loop vectorization (LV), and multicore parallelization using OpenMP (OMP). The Orio implementations of these transformations are described in [7].

The set of possible parameter values used for tuning directives is not comprehensive. We used manually selected values that are not dictated by any particular machine. However, the search space can be improved (reduced) by more careful selection of these parameters. SPAPT will evolve to take into account architectural features as it is used.

SPAPT is intended to be used for evaluating the search approaches in any autotuning system. We use Orio [7] as an initial framework because it is open source, flexible, easy to use, and provides a large number of transformations. It takes an *Orio-annotated C* or Fortran implementation of a problem as input, generates multiple transformed code variants of the annotated code, empirically evaluates the performance of the generated codes, and has the ability to select the best-performing code variant using some popular heuristic search algorithms. Orio annotations consist of semantic comments that encode the computation. A separate tuning specification contains various parameterized performance-tuning directives and sizes of inputs to consider. In addition to the general-purpose tuning directives such as UJ, CT, RT, SR, AC, LV, and OMP, Orio supports a number of machine-specific optimizations (e.g., generating calls to SIMD intrinsics on Intel and Blue Gene/P architectures). We refer the reader to [7] for a detailed account on annotation parsing and code generation schemes in Orio.

By integrating SPAPT with Orio we provide an immediate demonstration of its use and enable future use by other autotuning packages as interfaces to them are added during Orio development (Orio already interfaces to a number of third-party transformation and search tools and will continue to add more). However, the defined mathematical optimization problems in SPAPT are not Orio-specific and can be reimplemented in any other framework that supports the discussed transformations.

In Table 1, the column $|\mathcal{D}|$ shows, for each kernel, the number of feasible decision points, which ranges between $1.01e + 08$ and $2.73e + 30$. SPAPT is made available for download with Orio. Readers can also browse the benchmark set at <http://trac.mcs.anl.gov/projects/performance/browser/orio/testsuite/SPAPT.v.01>.

4 Illustrative Experiments

In this section, we present an illustrative experimental study using several problems from the benchmark set. Based on the results of this study, we discuss some of the characteristics of problems in SPAPT that are relevant for autotuning.

Experiments are carried out on dedicated nodes of a cluster in which each node contains two Intel Nehalem series quad-core 2.53 GHz processors, 64 KB L1 cache/core, 256 KB L2 cache/core, and 36 GB of memory running the stock Linux kernel version 2.6.18 provided by RedHat.

4.1 Effect of cache misses and the impact of performance metric choice

When a code is transformed and compiled with respect to a given parameter configuration, typically it has to be run on the target machine a number of times to overcome variations resulting from factors such as operating system noise and compulsory, capacity, and conflict

Table 2: Estimated mean and standard deviation of the runtime for 35 runs at the initial parameter configuration.

Problem	ADI	ATAX	BiCG	COR	COV	DGEMV	FDTD4d2d	GEMVER	GESUMMV
$\hat{\mu}_{init}$	0.5830	0.0052	0.0040	0.0009	0.0007	0.2697	0.7751	0.0742	0.0260
$\hat{\sigma}_{init}$	2.52e-05	4.90e-05	1.52e-05	8.93e-06	1.95e-06	4.30e-03	2.00e-04	7.83e-05	8.61e-05
Problem	Hessian	Jacobi-1d	LU	MM	MVT	Seidel	Stencil3d	Tensor	TRMM
$\hat{\mu}_{init}$	0.1679	0.0004	0.6969	0.0114	0.0009	0.2286	0.0992	0.2696	0.2252
$\hat{\sigma}_{init}$	2.78e-05	5.16e-06	8.27e-04	2.13e-05	3.00e-05	4.62e-05	4.11e-05	3.11e-04	1.65e-04

cache misses. Hence, modeling decisions related to the performance objective can play a significant role in the tuning process, in particular, when we have *a priori* knowledge on the data access patterns of the given application. In SPAPT we intentionally do not specify a fixed form of the objective because it can depend heavily on the target machine and the choice of the performance metric (e.g., runtime, flops, or power). Also, we target problems on which no *a priori* knowledge is available or all the knowledge on access patterns are exploited and there is still room for performance improvements. On the other hand, *a priori* knowledge can benefit performance objective modeling substantially.

In our exploratory studies, we consider minimizing the runtime for each problem. Many performance metrics can serve as an optimization objective in (1), including

$$f(x) = \frac{1}{m} \sum_{i=1}^m r_i(x), \quad f(x) = \text{median}_{i=1, \dots, m} r_i(x), \quad f(x) = \min_{i=1, \dots, m} r_i(x), \quad f(x) = r_3(x),$$

where $\{r_1(x), \dots, r_m(x)\}$ denote a sequence of m runtime realizations (replications) for parameter configuration x , and these objectives denote the mean, median, minimum, and third realized time, respectively. Performance objectives other than the mean, including those given above and quantile-based metrics, can be adopted based on the ultimate goals of the performance tuning process.

Next we discuss various considerations related to performance objectives given $m = 35$ consecutive replications, without flushing the data from cache, for each run. The sample mean runtime is often used to approximate uniform system conditions because it can asymptotically reduce nondeterministic variations in the runs. In Table 2, we show the sample mean $\hat{\mu}_{init}$ and standard deviation $\hat{\sigma}_{init}$ of the runtime for 35 runs at the initial parameter configuration for some problems with input size N . The mean is stable to three or four significant digits considering the relative noise ($\hat{\sigma}_{init}/\sqrt{35}\hat{\mu}_{init}$).

Figures 1(a) and 1(b) illustrate a comparison of the mean, median, minimum, and third runtime values of 5,000 random parameter configurations in $|\mathcal{D}|$. Note that all the configurations in the horizontal axis are sorted with respect to the mean, so that the mean is monotone increasing. The results show that in a large number of parameter configurations from ATAX.01.N (Figure 1(a)), the median, minimum, and third runtime differ significantly

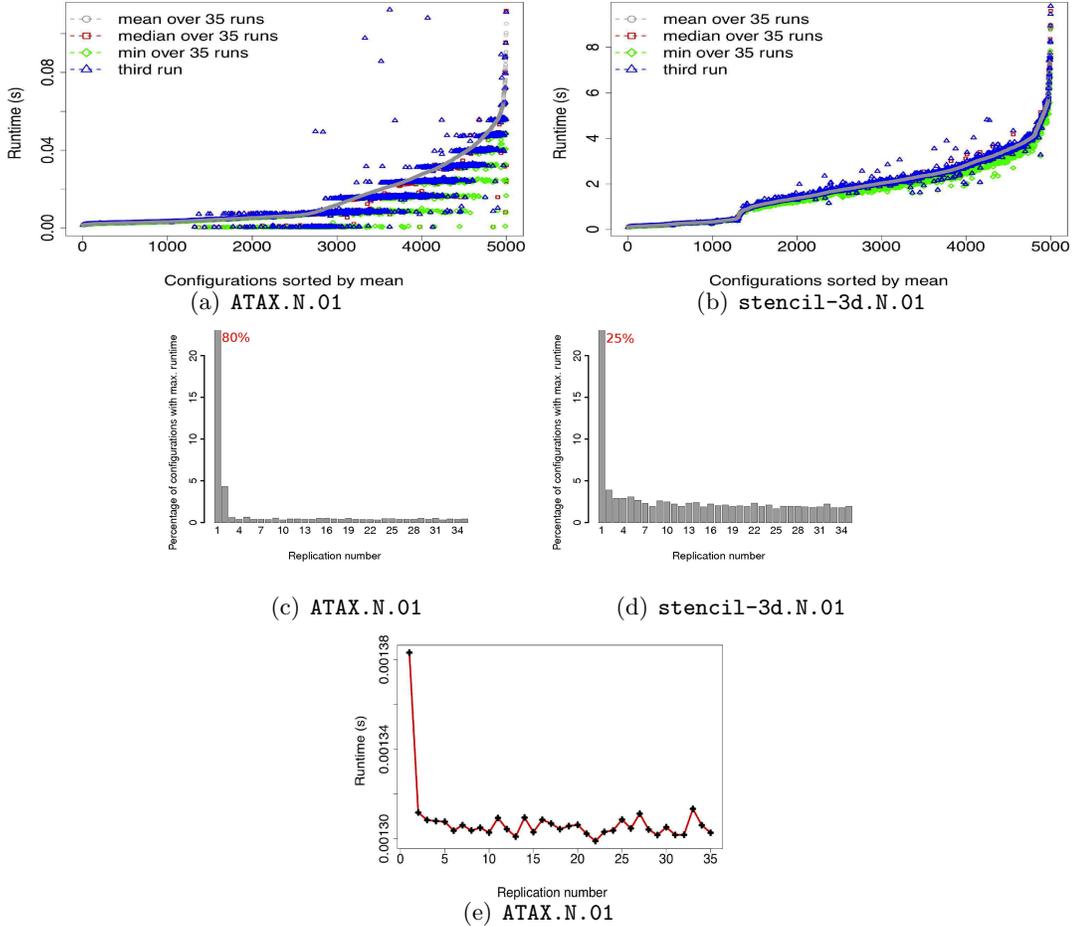


Figure 1: Comparison of performance objectives and the effect of cache misses in the SPAPT problems.

from the mean. However, these metrics are similar to each other on the results for the problem `stencil-3d.01.N` (Figure 1(b)). Since the `ATAX` and `stencil3d` kernels are memory- and computation-bound, respectively, the former is more sensitive to cache misses than the latter. Figures 1(c) and 1(d) show the percentage of configurations with maximum runtime for each replication number. For `ATAX.01.N`, in 80% of 5,000 configurations, the first run has the maximum runtime, whereas in `stencil3d` this drops to 25%. Figure 1(e) shows the runtime realizations as a function of the replication number for the initial configuration of `ATAX.01.N`. As expected, the execution times of the first few runs are longer than those of the other runs. Note that the performance objective of the third runtime value is explicitly designed to take this into account. We observed that the trend of results from `ADI.01.N`, `BiCG.N.01`, `COR.N.01`, `COV.N.01`, `GEMVER.N.01`, `Jacobi.N.01`, `MVT.N.01`, `Seidel.N.01`, `Tensor.N.01`, and `TRMM.N.01` is similar to `ATAX.01.N` and others are similar to `Stencil-3d` (see the online appendix [29]).

From the modeling perspective, these results imply that when a kernel is highly sensitive to cache misses, one has to be careful choosing the performance objective. Inside an application, if the data required for a particular kernel is normally not present in cache when

the kernel is executed, the tuning process must reflect this by flushing the cache for each replication. On the other hand, even if the kernel is highly sensitive to cache misses but it is known that the required data is present in cache when the kernel is invoked, then we must ignore first few repetitions during tuning. Further, when the kernel is compute-bound and not sensitive to cache misses, tuning with a large number of repetitions results in a waste of resources. In such cases, the third runtime value is a good choice. In the rest of this section, we use the widely adopted mean runtime (in 35 replications) as the performance metric.

In this experimental setup, the impact of secondary effects such as branch predictions, improved fetch, load aliasing, and instruction decoding and its related consequence are not included in the performance metric. The Orio framework that we use does not support removal of the secondary effects beyond the ability to control the number of replications.

4.2 Impact of the target machine

We now analyze the impact of different machines on the mean runtime of the parameter configurations from SPAPT problems. In addition to the Intel Nehalem cluster, we use two large-scale leadership computing machines: IBM Blue Gene/P and Cray XE6. Each node of the Blue Gene/P contains IBM PowerPC 850 MHz quad-core processors with 32 KB L1 cache, 4 128 byte-line buffers L2 cache, 8 MB L3 cache, and 2 GB of memory running Compute Node Kernel OS. Each node of the Cray XE6 contains 2 twelve-core AMD MagnyCours 2.1 GHz processors with 64 KB L1 cache, 512 KB L2 cache, 6 MB L3 cache, and 32 GB of memory running Cray Linux Environment OS.

Figure 2(a) shows the mean runtime correlation between the IBM Blue Gene/P and Intel Nehalem cluster for configurations from `ATAX.N.01`. We observe that high-performing parameter configurations for the the Intel Nehalem cluster (mean runtime between 0.001 and 0.005 seconds) obtain poor mean runtimes (between 0.02 and 0.04 seconds) on the IBM Blue Gene/P and vice versa. We found that enabling OpenMP in the Intel Nehalem nodes degrades the performance of the code because of the OpenMP overhead. On the IBM Blue Gene/P, however, it leads to performance improvements because this machine has slower processors and a smaller cache size and less memory per core. The two distinct clusters of configurations in Figure 2(a) correspond to the codes with OpenMP enabled and disabled. Nevertheless, the Intel Nehalem cluster is closer to the Cray XE6 in terms of computing power and memory. From Figure 2(b), we can observe that the mean runtime of the parameter configurations run on the Intel Nehalem cluster and the Cray XE6 exhibit high correlation. From the results, we expect that generalization of parameter configurations depends on the kernel and the target machines.

4.3 Performance objective density

A naive way to assess the difficulty of an optimization problem in SPAPT consists of sampling parameter configurations at random and measuring the density of their performance objectives. Figure 3 shows histograms of the objective values obtained on 5,000 random pa-

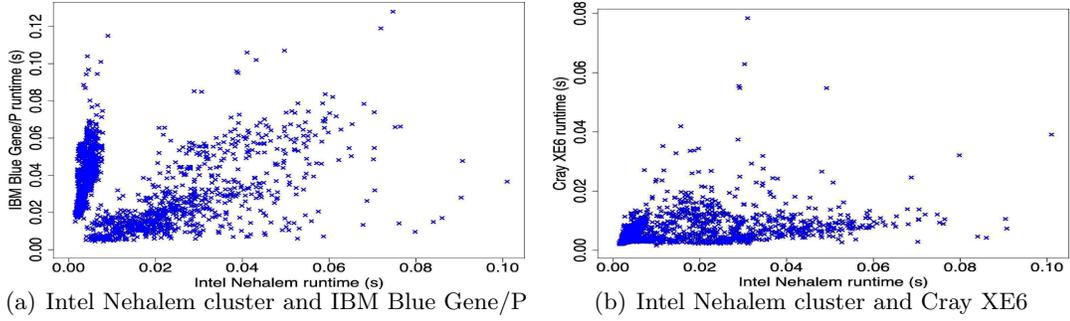


Figure 2: Mean runtime correlation between the configurations on the SPAPT problem ATAX.01.N.

parameter configurations on different problems from SPAPT. We observe that for ADI.N.01 and FDTD4d2d.N.01, the number of high-performing parameter configurations is low compared with that for BiCG.N.01 and Seidel.01.N. We expect that a simple random search can find high-performing configurations in short computation time for BiCG.N.01 and Seidel.01.N, whereas ADI.01.N and FDTD4d2d.N.01 might require sophisticated search algorithms. The performance objective density plots for other problems are given in the online appendix [29]. Given the large search space of the optimization problems and the number of random parameter configurations considered, the density results should be treated as baseline results; they should not be taken as an exhaustive metric for assessing the difficulty of solving a particular search problem in the benchmark.

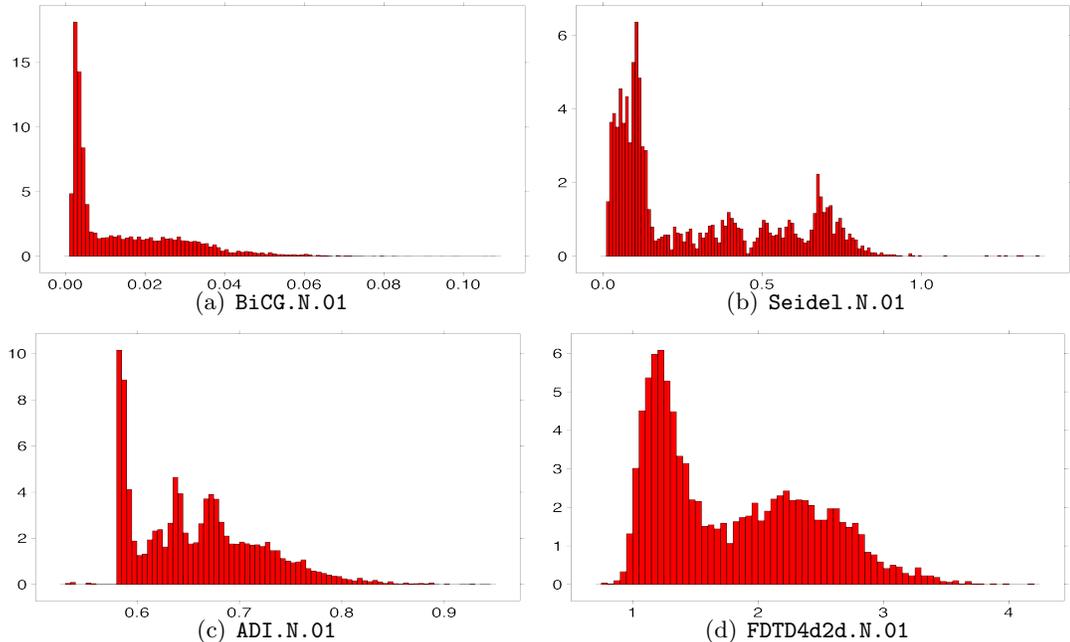


Figure 3: Illustrative histogram of mean runtime from 5,000 random code variants in \mathcal{D} on SPAPT problems.

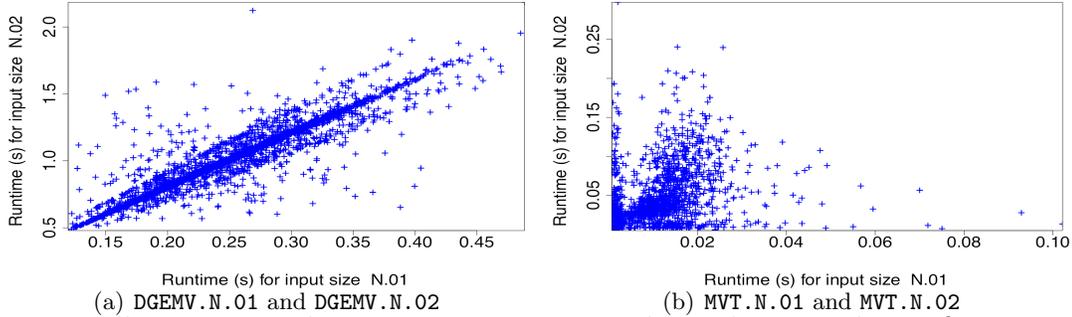


Figure 4: Illustrative results on mean runtime correlation between the configurations on SPAPT problems.

4.4 Impact of input size

Another factor that plays a crucial role in autotuning is the size of the arrays involved in the computation. In most cases, tuning has to be performed for a number of different input sizes because the best parameter configuration obtained for one input size is not necessarily the best for a different input size. In some cases, however, parameter configurations can be generalized. This is illustrated in Figures 4(a) and 4(b), which show the correlation between the objectives for different input sizes for two kernels. In problems based on the MVT kernel (Figure 4(b)), a large number of high-performing parameter configurations for input size $N.01$ become less effective for input size $N.02$. This result occurs because transformations targeting different levels of the memory hierarchy would not produce the same effect on a computation that can fit in registers or L1 as they would on an instance that does not fit in any level of cache. Nevertheless, the results for problems based on the DGEMV kernel (Figure 4(a)) show that high-performing parameter configurations are generalizable for certain types of computations. We also observe some correlations in `ADI`, `GESUMMV`, and `Stencil3d` (see [29]).

5 Conclusions and Future Directions

Motivated by a lack of a test suite of search problems in autotuning, we developed SPAPT. Each problem in SPAPT is a well-defined mathematical optimization problem based on a representative kernel from a scientific application, parameterized tuning directives, acceptable values for each parameter, input sizes, and an initial configuration for search algorithms. To the best of our knowledge, SPAPT is the first test suite in the autotuning literature that is designed for analyzing and benchmarking mathematical optimization algorithms. We implemented all these problems in an annotation-based language that can be processed by `Orio`, a recently developed performance tuning software framework. We conducted experiments to show performance impacts of problem characteristics such as choice of performance objectives, noise, effect of cache misses, target machines, and input sizes.

SPAPT has the potential to improve the state of the art in autotuning. On the one hand, our easily accessible, portable `Orio` implementation of the test suite can encourage mathematical optimization researchers to develop optimization algorithms without know-

ing the fine details of compiler optimization and performance tuning. On the other hand, the autotuning community will benefit from better algorithms and can use SPAPT to conduct systematic experimental studies of the existing optimization algorithms and better understand the role that different transformations play.

In addition to the limitations of any test suite described in Section 1, SPAPT has the following limitations at present. It deals only with codes that run on a single node and does not provide any codes that run on parallel machines. In future suites, we plan to include sparse matrix kernels, parallel codes, and kernels from other well-known benchmarks such as TORCH. Moreover, we will extend the application space and numerical and scientific problem domain coverage of the test suite. We used only the set of parameterized code transformations supported by Orio. While these transformations are highly relevant for single-node performance, distributed-memory, parallel codes demand a different set of transformations. Both SPAPT and Orio will evolve taking this into account. We will use SPAPT to understand the search problem characteristics, to benchmark existing optimization algorithms, and to develop efficient optimization algorithms for autotuning. We will investigate further the impact of different target machines on the performance objectives of the SPAPT problems. We also intend to build a database of tabulated execution times to further facilitate benchmarking of search algorithms.

References

- [1] D. Bailey, R. Lucas, S. Williams (Eds.), *Performance Tuning of Scientific Applications*, Chapman & Hall/CRC Computational Science, 2010.
- [2] P. Balaprakash, S. Wild, P. Hovland, Can search algorithms save large-scale automatic performance tuning?, in: *The International Conference on Computational Science*, 2011.
- [3] 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.
- [4] J. J. Moré, B. S. Garbow, K. E. Hillstom, Testing unconstrained optimization software, *ACM Transactions on Mathematical Software* 7 (1) (1981) 17–41. doi:http://doi.acm.org/10.1145/355934.355936.
- [5] N. I. M. Gould, D. Orban, P. L. Toint, CUTEr and SifDec: A constrained and unconstrained testing environment, revisited, *ACM Transactions on Mathematical Software* 29 (4) (2003) 373–394. doi:http://doi.acm.org/10.1145/962437.962439.
- [6] J. J. Moré, S. M. Wild, Benchmarking derivative-free optimization algorithms, *SIAM Journal on Optimization* 20 (1) (2009) 172–191.
- [7] 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.
- [8] T. Kisuki, P. M. W. Knijnenburg, M. F. P. O’Boyle, Combined selection of tile sizes and unroll factors using iterative compilation, in: *Proc. of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, Washington, DC, 2000.
- [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] K. Seymour, H. You, J. Dongarra, A comparison of search heuristics for empirical code optimization, in: *Proc. of the 2008 IEEE International Conference on Cluster Computing*, 2008, pp. 421–429.
- [11] 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: *Proc. of the Fourth International Workshop on Automatic Performance Tuning*, Japan, 2009.
- [12] A. Tiwari, C. Chen, C. Jacqueline, M. Hall, J. K. Hollingsworth, A scalable autotuning framework for compiler optimization, in: *Proc. of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, Washington, DC, 2009, pp. 1–12.

- [13] L.-N. Pouchet, PolyBench: The Polyhedral Benchmark suite (2011).
URL <http://www-roc.inria.fr/~pouchet/software/polybench/>
- [14] A. Kaiser, S. Williams, K. Madduri, K. Ibrahim, D. Bailey, J. Demmel, E. Strohmaier, TORCH computational reference kernels: A testbed for computer science research, Tech. Rep. UCB/EECS-2010-144, EECS Department, University of California, Berkeley (December 2010).
URL <http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-144.html>
- [15] The Embedded Microprocessor Benchmark Consortium, <http://www.eembc.org>.
- [16] HPC Challenge benchmark, <http://icl.cs.utk.edu/hpcc/>.
- [17] The Parboil benchmark suite, <http://impact.crhc.illinois.edu/parboil.php>.
- [18] SPEC benchmarks, <http://www.spec.org/benchmarks.html>.
- [19] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrisnan, S. Weeratunga, The NAS Parallel Benchmarks, *International Journal of High Performance Computing Applications* 5 (3) (1991) 63–73.
- [20] C. Bienia, S. Kumar, J. P. Singh, K. Li, The PARSEC benchmark suite: Characterization and architectural implications, in: *Proceedings of the 17th international conference on Parallel architectures and compilation techniques, PACT '08*, ACM, 2008, pp. 72–81.
URL <http://doi.acm.org/10.1145/1454115.1454128>
- [21] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, K. Skadron, Rodinia: A benchmark suite for heterogeneous computing, in: *IEEE International Symposium on Workload Characterization, 2009. IISWC 2009.*, 2009, pp. 44–54.
- [22] J. Dongarra, P. Luszczek, A. Petitet, The LINPACK benchmark: Past, present and future, *Concurrency and Computation: Practice and Experience* 15 (9) (2003) 803–820.
- [23] J. D. McCalpin, Stream: Sustainable memory bandwidth in high performance computers, Tech. rep., University of Virginia, Charlottesville, Virginia (1991-2007).
URL <http://www.cs.virginia.edu/stream/>
- [24] C. C. Minh, J. Chung, C. Kozyrakis, K. Olukotun, STAMP: Stanford Transactional Applications for Multi-Processing, in: *IISWC '08: Proc. of the IEEE International Symposium on Workload Characterization, Seattle, WA, 2008*, pp. 35–46.
- [25] J. P. Singh, W.-D. Weber, A. Gupta, SPLASH: Stanford parallel applications for shared-memory, *SIGARCH Comput. Archit. News* 20 (1992) 5–44.
- [26] The pChase Memory Benchmark Page, <http://pchase.org/>.

- [27] J. Demmel, J. Dongarra, A. Fox, S. Williams, V. Volkov, K. Yelick, Accelerating time-to-solution for computational science and engineering, *SciDAC Review* (15).
- [28] S. Kamil, C. Chan, L. Oliker, J. Shalf, S. Williams, An auto-tuning framework for parallel multicore stencil computations, in: *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2010, pp. 1–12.
- [29] P. Balaprakash, S. Wild, B. Norris, Supplementary page for SPAPT: Search problems in automatic performance tuning (2012).
URL <http://www.mcs.anl.gov/~pbalapra/supp/SPAPT/supplementary.html>

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.