

SPAPT: Search Problems in Automatic Performance Tuning

Prasanna Balaprakash Stefan M. Wild Boyana Norris

Mathematics and Computer Science Division
Argonne National Laboratory
9700 South Cass Avenue
Argonne, IL 60439
{pbalapra,wild,norris}@mcs.anl.gov

Abstract

Automatic performance tuning of computationally intensive kernels in scientific applications is a promising approach to achieving good performance on different computing architectures 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 a number of problems from the test suite. We discuss important issues such as modeling, search space characteristics, and performance objectives.

Keywords autotuning, performance tuning, benchmark, test suite, kernels

1. Introduction

The landscape of scientific application programming is undergoing rapid changes as a result of increasingly complex computing architectures and the quest for high performance on these architectures. Chasing performance gains through manual tuning becomes a complex and time-consuming process that is neither scalable nor portable. Automatic performance tuning (in short, *auto-tuning*), or empirical performance tuning, is a promising and viable approach to address the limitations of manual tuning. Auto-tuning involves three major phases: identifying code optimization

techniques that are relevant to the given code and architecture, 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 architecture. In recent years, this has emerged as an effective approach to tune scientific kernels for both serial and multicore processors [10, 11, 18, 19, 32, 33].

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 [7], the authors showed that the search problem arising in autotuning can be formulated as a mathematical optimization problem and 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 performance problems and subsequently transforming, compiling, and running the corresponding codes. In fact, recent successes of performance tuning in mathematical optimization have focused on obtaining parameters for other optimization algorithms (e.g., [5]), these codes being what optimizers are most familiar with.

On the other hand, a rich history in mathematical optimization of sets of benchmark problems exists. Examples include the Moré-Garbow-Hillstom problems for unconstrained optimization [23]; the more general CUTer set [14] (a subset of which was used as the inputs in [5]); and the smooth, noisy, and nonsmooth problems in [24]. 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 problems only that show favorable aspects of an 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 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 number of lower-level, serial performance tuning tasks in scientific applications. In particular, we focus on kernels in scientific codes. We implement problems in a format that can be readily processed by Orio [16, 25], a recently developed empirical-performance tuning software framework. By making Orio explicitly part of the set and defining specific search problems, our first goal is to attract the mathematical optimization community to help advance the field of performance tuning. 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 testable test suite from the perspective of the mathematical optimization community. We did not want to restrict its applicability to parallel codes on large computational clusters and/or leadership-class machines because the mathematical optimization researchers might have limited access to these machines. However, given that most of the single-node machines—including desktops and laptops—come with multi-cores, search problems in SPAPT contain OpenMP directives as code transformation techniques. Second, single-node performance tuning is highly relevant in a number of kernels where the communication cost between the processor and the memory hierarchy is a bottleneck for the performance.

The rest of the paper is organized as follows. In Section 2 we review related work on test suites for autotuning. In Section 3 we give a high-level overview of SPAPT. We briefly give an account on each class of kernels, application context, and tunable parameters. In Section 4, using an illustrative experimental study on a number of problems from the benchmark, we discuss some important issues related to modeling, optimization, and performance objectives.

2. Related Work

Balaprakash et al. [7], Kisuki et al. [20], Qasem et al. [27], Seymour et al. [28], Shin et al. [29], and Tiwari et al. [32] used a number of linear algebra kernels for autotuning. Pouchet [26] 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. [25] and Hartono et al. [16] used a collection of linear algebra kernels, solvers, and stencils. These are parameterized codes that were used to test the effectiveness of Orio. 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. [17] proposed the TORCH testbed, a set of reference kernels to enable software and hardware co design. 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. [17] argue that a number of existing test suites can be seen as reference implementations of one or more kernels from TORCH. Examples include EEMBC [1], HPC Challenge

[2], ParBoil [3], SPEC [4], NAS Parallel test suites [6], PARSEC [8], Rodinia [9], LINPACK [13], STREAM [21], STAMP [22], SPLASH [30], and pChase [31]. Although in principle these test suites can be parameterized and used for autotuning, none of them are developed specifically for evaluating the performance of the search problem in autotuning. Hence, there is a noticeable void in the literature of test suite sets of well-formulated search problems in autotuning.

The SPAPT set that we propose in this paper is based on [16, 17, 25, 26] and comprises representative examples from autotuning in scientific applications. 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 as a transparent entity—one can easily integrate an optimization algorithm to tackle the search problem without knowing the nitty-gritty details related to the code transformation techniques, compiler specifics, and the target architecture. The particularity of a search problem in SPAPT is that it is a well-defined mathematical optimization problem composed of a 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.

3. Test Suite

In this section we provide a high-level overview of the test suite and the chosen tuning directives. We then discuss their implementations in Orio.

3.1 Reference kernels and search problems

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 can result in significant overall application performance improvements [12]. A range of transformations can be applied leading to better utilization of the memory hierarchy and aiding in exploiting shared-memory parallelism on multicore architectures. The SPAPT benchmark that we propose in this paper comprises 18 such kernels. These kernels are grouped into four groups as in [26]: linear algebra computation kernels, linear algebra solver kernels, stencil code kernels, and data-mining kernels.

Linear algebra computation kernels. These 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 involve elementary linear algebra operations such as vector/matrix/tensor multiplications and transposes. See Table 1 for a summary of the operations involved.

Linear algebra solver kernels. Linear algebra solvers 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. Stencil codes follow a regular pattern to access and update array elements. They are commonly used in implicitly and explicitly solving partial differential equations [18]. In this group, we have four kernels from ADI preconditioners (ADI), Jacobi 1-D (Jacobi-1d), Seidel stencil (Seidel), and 3-D stencils computations (Stenci13d).

Data mining kernels. In this group, we have two kernels: correlation (COR) and covariance (COV) computations. They involve finding statistical relationships among a number of random vari-

Table 1. Collection of test suite kernels.

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

ables, which is central to many statistical packages. The reference implementations are obtained from [26].

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 architecture and a single performance objective f , both discussed further in Section 4, this search problem is equivalent to the mathematical optimization problem

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

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 [7]. 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 bounded. 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. First are algebraic constraints, where the time required to verify the feasibility ($x \in \mathcal{D}$) of an arbitrary point $x \in \mathcal{R}^n$ 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 $RT_I * RT_J \leq 150$. Second are 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 as it can help the optimization algorithm move away from regions of infeasibility.

Hidden constraints. These constraints are attributed to unsuccessful code evaluations that occur due to transformation, compilation, and run-time errors. While failure at the code transformation phase is relatively cheap, failure due to run-time 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.01.N, ATAX.02.N, ATAX.04.N, and ATAX.01.N.nb. The naming conventions take the following meaning: N is the (reference) input size in ATAX.01.N; $2 \times N$ and $4 \times N$ are the input sizes in ATAX.02.N and ATAX.04.N, 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.01.N.nb is obtained from ATAX.01.N 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 facilitate adoption of advanced 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 implementation without any code transformation and optimization. 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., binary decisions such as enabling or disabling OpenMP create discontinuities in the search space).

Table 1 gives a high-level overview of the kernels and tuning transformations considered for each kernel. Whenever applicable, we adopt the following general-purpose, parameterized tuning directives: loop unroll/jamming (UJ), cache tiling (CT), register tiling (RT), loop permutation (LPM), 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 [15, 16].

3.2 Orio-specific implementations

Orio [16, 25] is a recently developed extensible and portable software framework for empirical performance tuning. 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, LPM, SR, AC, LV, and OMP, Orio supports a number of architecture-specific optimizations (e.g., generating calls to SIMD intrinsics on Intel and Blue Gene/P architectures). We refer the reader to [16, 25] for a detailed account on annotation parsing and code generation schemes in Orio.

SPAPT is intended to be used for evaluating the search approaches in any autotuning system. By integrating it 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).

From an optimization perspective, for a given search problem, one needs to know the tunable parameters, possible values for each parameter, and a starting parameter configuration. A concrete annotation example is shown in Figure 1. Note that for brevity, we skip other important regions of the annotation such as the tuning directives, kernel, and compiler options in the annotation. The example shows tunable performance parameters for CT, AC, UJ, SR, LV, and OMP; their possible values together with the constraints on CT and UJ; and the input size. 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 at trac.mcs.anl.gov/projects/performance/wiki/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 on several problems from the benchmark set. We use the results of this study to discuss some of the characteristics of problems in SPAPT that are highly relevant for autotuning.

Experiments are carried out on dedicated nodes of the Fusion cluster at Argonne National Laboratory. Each node of Fusion 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 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

```
def performance_params
{
# Cache tiling
param T1_I[] = [1,16,32,64,128,256,512];
param T1_J[] = [1,16,32,64,128,256,512];
param T2_I[] = [1,64,128,256,512,1024,2048];
param T2_J[] = [1,64,128,256,512,1024,2048];

# Array-copy
param ACPY_A[] = [False,True];

# Unroll-jam
param U_I[] = range(1,31);
param U_J[] = range(1,31);

# Scalar replacement
param SCREP[] = [False,True];

# Loop Vectorization
param VEC[] = [False,True];

# Parallelization
param OMP[] = [False,True];

# Constraints
constraint tileI = ((T2_I == 1)
or (T2_I % T1_I == 0));
constraint tileJ = ((T2_J == 1)
or (T2_J % T1_J == 0));
constraint reg_capacity = (2*U_I*U_J +
2*U_I + 2*U_J <= 130);
}

let SIZE = 1000;

def input_params
{
param MSIZE = SIZE;
param NSIZE = SIZE;
param M = SIZE;
param N = SIZE;
}
/*@ begin Loop(
/* transform module */
/* kernel code */
) @*/
/*@ end @*/
```

Figure 1. Parameter specification and constraint example in Orio.

heavily on the target architecture and the choice of the performance metric (e.g., runtime, flops, or power).

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

$$f(x) = \frac{1}{m} \sum_{i=1}^m r_i(x),$$

$$f(x) = \text{median}_{i=1,\dots,m} r_i(x),$$

$$f(x) = \min_{i=1,\dots,m} r_i(x),$$

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

Problem	$\hat{\mu}_{init}$	$\hat{\sigma}_{init}$
ATAX	0.0052	2.05e-05
BiCG	0.0040	5.68e-06
COR	0.0009	1.62e-06
DGEMV	0.0100	5.33e-06
GEMVER	0.0328	3.71e-04
GESUMMV	0.0259	4.54e-05
Jacobi	0.0004	1.45e-06
MM	0.0211	3.58e-06
MVT	0.0017	7.18e-06

$$f(x) = r_3(x),$$

where $\{r_1(x), \dots, r_m(x)\}$ denote a sequence of m runtime realizations 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 ($\hat{\sigma}_{init}/\sqrt{35}\hat{\mu}_{init}$).

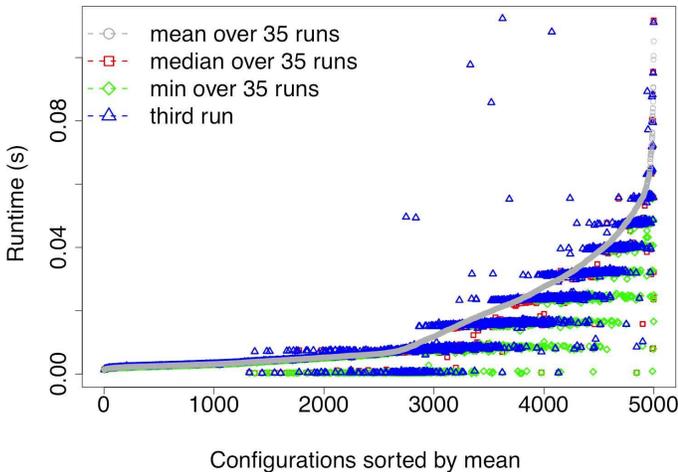


Figure 2. Comparison of performance objectives on the SPAPT problem ATAX.01.N.

Figures 2 and 3 show 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 x 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 2), the median, minimum, and third runtime differ significantly from the mean. However, these metrics are similar to each other on the results for the problem stencil-3d.01.N (Figure 3). Since ATAX and stencil3d kernels are memory- and computation-bound, respectively, the former is more sensitive to cache misses than the latter. Figures 4

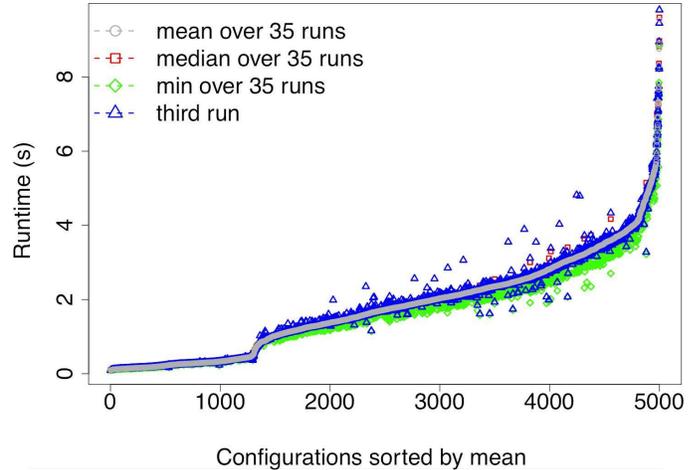


Figure 3. Comparison of performance objectives on the SPAPT problem stencil-3d.01.N.

and 5 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 6 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.

From the modeling perspective, these results imply that when a kernel is highly sensitive to cache misses, one has to be careful with the choice of 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 30 replications) as the performance metric.

4.2 Impact of the target machine

We now analyze the impact of different architectures on the mean runtime of the parameter configurations from SPAPT problems. In addition to Fusion, we use two large-scale leadership computing machines: Intrepid (IBM Blue Gene/P) from Argonne National Laboratory and Hopper (Cray XE6) from the National Energy Research Scientific Computing Center. Each node of Intrepid 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 Hopper 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 7 shows the mean runtime correlation between the configurations from ATAX.01.N on Intrepid and Fusion. We observe that high-performing parameter configurations for Fusion (mean runtime between 0.001 and 0.005 seconds) obtain poor mean runtimes (between 0.02 and 0.04 seconds) on Intrepid and vice versa. We found that enabling OpenMP in Fusion degrades the performance of the code because of the OpenMP overhead. However, in

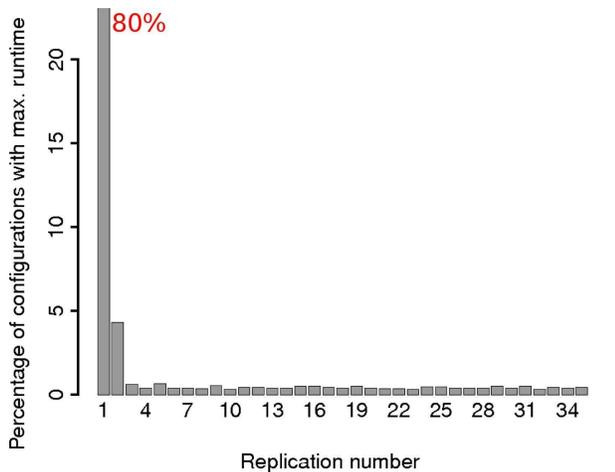


Figure 4. Effect of cache misses on the SPAPT problem ATAX.01.N: percentage of configurations with maximum runtime as a function of replication number.

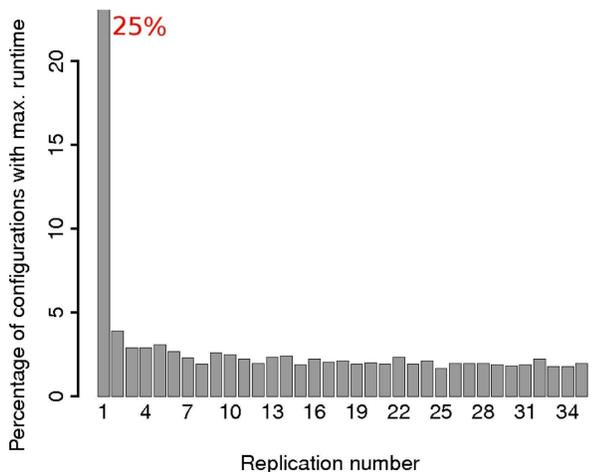


Figure 5. Effect of cache misses on the SPAPT problem stencil3d.01.N: percentage of configurations with maximum runtime as a function of replication number.

Intrepid it leads to performance improvements because Intrepid has slower processors and a smaller L1, L2, and memory per core. The two distinct clusters of configurations in Figure 7 correspond to the codes with OpenMP enabled and disabled. Nevertheless, Fusion is closer to Hopper in terms of computing power and memory. From Figure 8, we can observe that the mean runtime of the parameter configurations run on Fusion and Hopper exhibit high correlation.

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. In Figures 9, 10, 11, and 12, we show histograms of the objective values obtained on 5,000 random parameter configurations on different problems from the benchmark set. We observe that for ADI.01.N and DGEMV.01.N problems, the number of high-performing parameter configurations is low compared with that for the GEMVER.01.N and SEIDEL.01.N. We expect that a simple random search can find high-performing configurations for GEMVER.01.N and SEIDEL.01.N, for which there are many high-performing parameter configurations, whereas ADI.01.N and

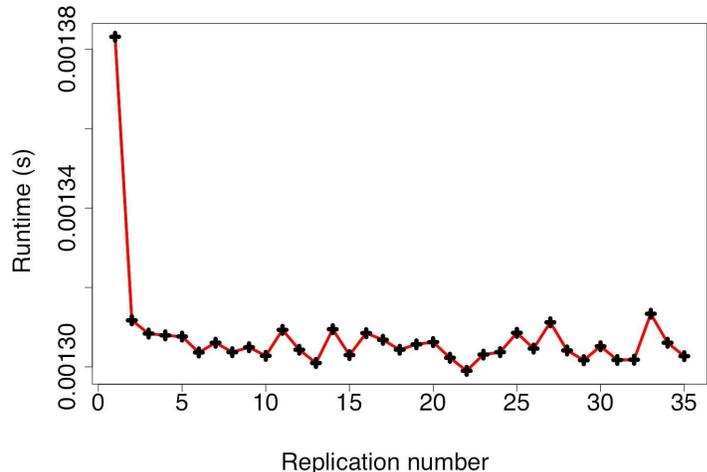


Figure 6. Effect of cache misses on the SPAPT problem ATAX.01.N: runtime realization as a function of replication number.

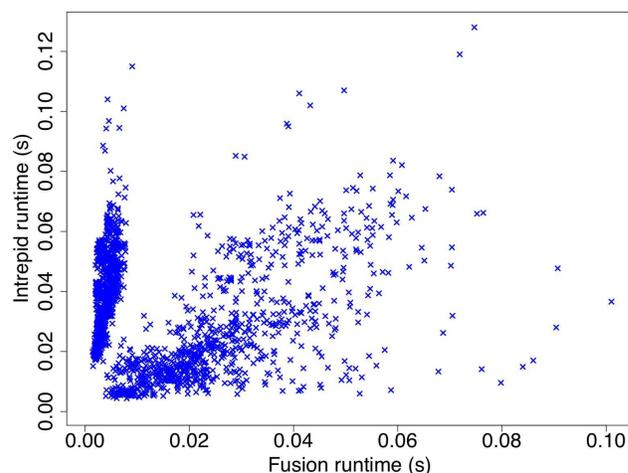


Figure 7. Mean runtime correlation between the configurations from Intrepid and Fusion on the SPAPT problem ATAX.01.N.

DGEMV.01.N problems might require sophisticated search algorithms. 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.

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 13 and 14, which shows the correlation between the objectives for different instance sizes. In problems based on the ATAX kernel (see Figure 13), a large number of high-performing parameter configurations for input size N become less effective for input size $4N$. 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,

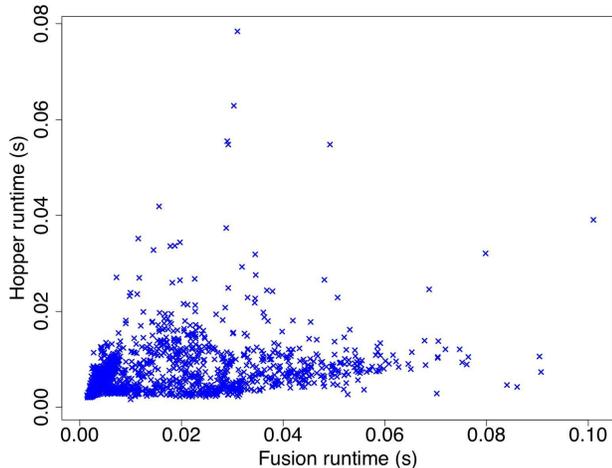


Figure 8. Mean runtime correlation between the configurations from Hopper and Fusion on the SPAPT problem ATAX.01.N.

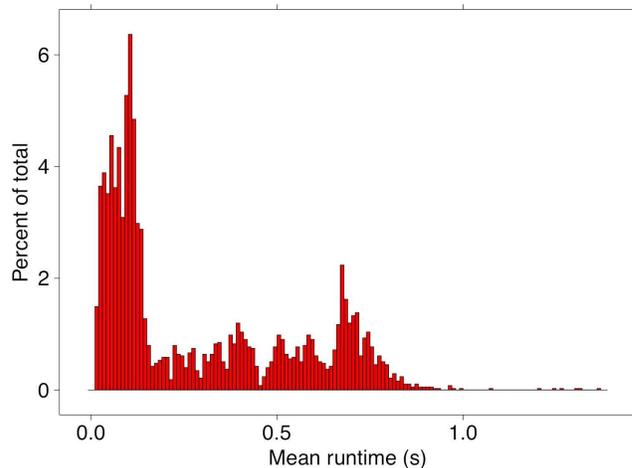


Figure 10. Histogram of mean runtime from 5,000 random code variants in \mathcal{D} on the SEIDEL.01.N problem.

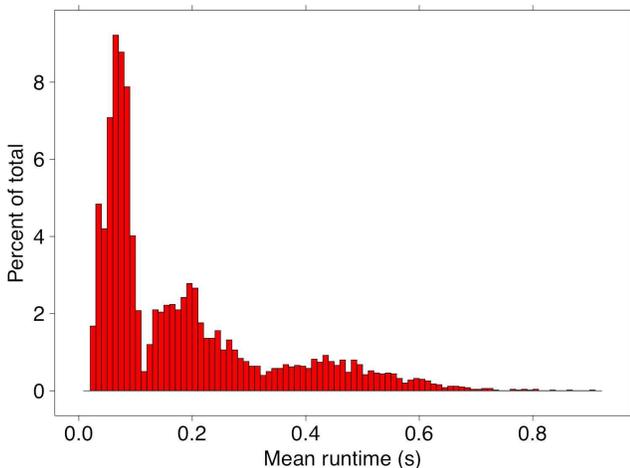


Figure 9. Histogram of mean runtime from 5,000 random code variants in \mathcal{D} on the GEMVER.01.N problem.

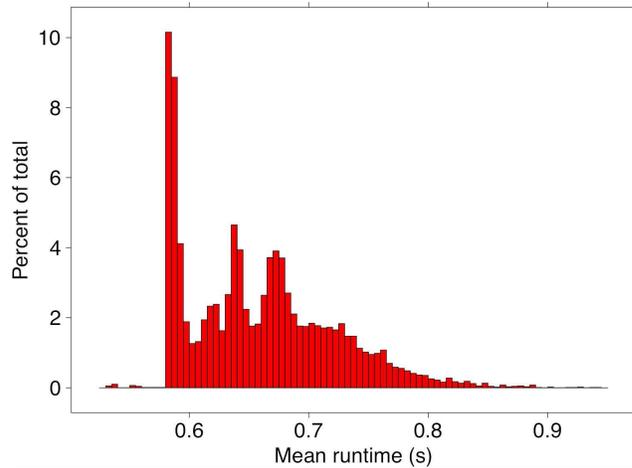


Figure 11. Histogram of mean runtime from 5,000 random code variants in \mathcal{D} on the ADI.01.N problem.

the results for problems based on the BiCG kernel (see Figure 14) show that high-performing parameter configurations are generalizable for certain types of computations.

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 illustrative 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 op-

timization researchers to develop optimization algorithms without knowing the fine details of compiler optimization and performance tuning. On the other hand, it can help the autotuning community 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. As a starting point, we focused on some of the widely used scientific kernels in the autotuning literature. A possible bias in the test suite is that a large number of problems deal with linear algebra-related computations. 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 different set of transformations.

We plan to continue to extend the application space and numerical and scientific problem domain coverage of the test suite. In particular, we will define search problems using additional kernels from TORCH. We will use SPAPT to understand the search problem characteristics, to benchmark existing optimization algorithms, and to develop efficient optimization algorithms for autotuning. We

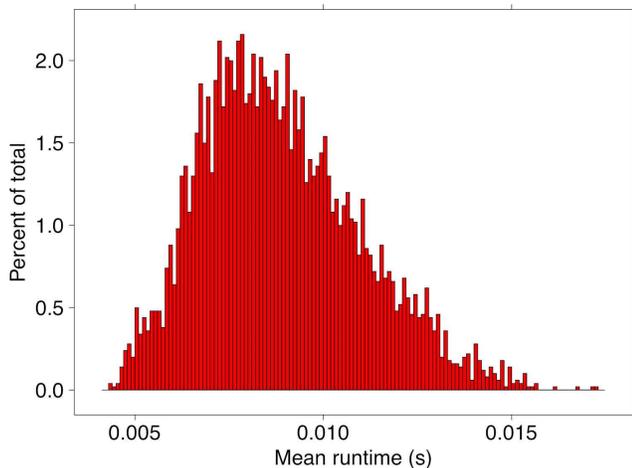


Figure 12. Histogram of mean runtime from 5,000 random code variants in \mathcal{D} on the DGEMV.01.N problem.

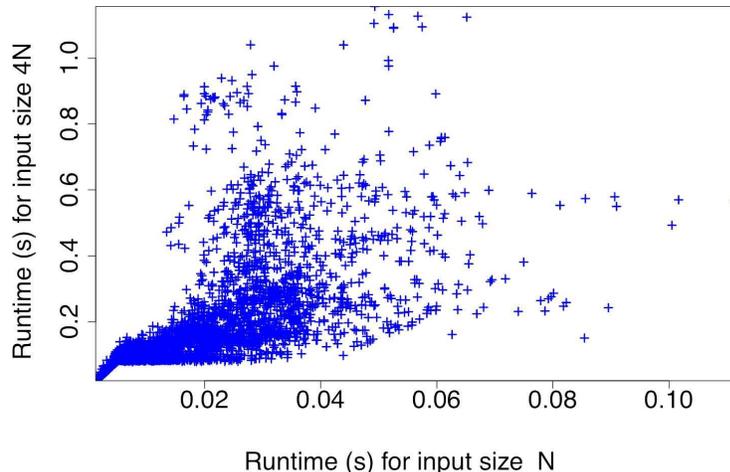


Figure 14. Mean runtime correlation between the configurations from the BiCG.01.N and BiCG.04.N problems.

Acknowledgments

This work was supported in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357. We are grateful to Paul D. Hovland for helpful discussions and to the Laboratory Computing Resource Center at Argonne National Laboratory.

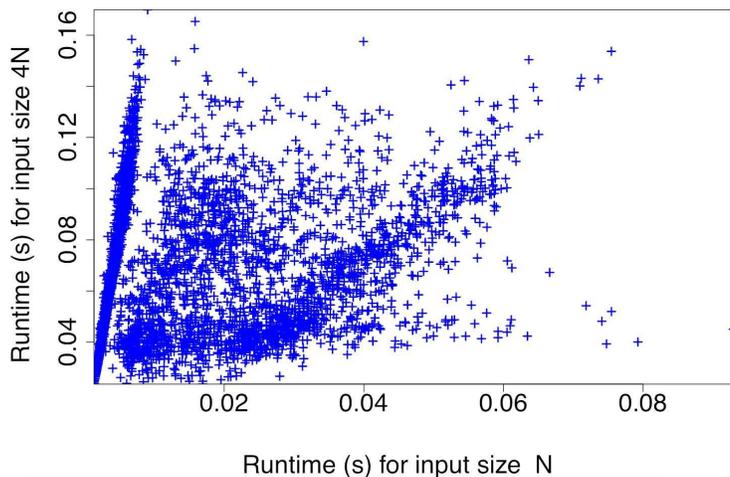


Figure 13. Mean runtime correlation between the configurations from the ATAX.01.N and ATAX.04.N problems.

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] The embedded microprocessor benchmark consortium. <http://www.eembc.org>.
- [2] HPC Challenge benchmark. <http://icl.cs.utk.edu/hpcc/>.
- [3] The Parboil benchmark suite. <http://impact.crhc.illinois.edu/parboil.php>.
- [4] SPEC benchmarks. <http://www.spec.org/benchmarks.html>.
- [5] C. Audet, D. C.-K, and D. Orban. Algorithmic parameter optimization of the DFO method with the OPAL framework. In K. Naono, K. Teranishi, J. Cavazos, and R. Suda, editors, *Software Automatic Tuning: From Concepts to State-of-the-Art Results*, pages 255–274. Springer, 2010.
- [6] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS Parallel Benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.
- [7] P. Balaprakash, S. Wild, and P. Hovland. Can search algorithms save large-scale automatic performance tuning? In *The International Conference on Computational Science*, July 2011.
- [8] C. Bienia, S. Kumar, J. P. Singh, and 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, pages 72–81. ACM, 2008.
- [9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization, 2009. IISWC 2009.*, pages 44–54, Oct. 2009.
- [10] I. Chung and J. Hollingsworth. A case study using automatic performance tuning for large-scale scientific programs. In *Proc. of Int. Symp. on High Performance Distributed Computing*, 2006.
- [11] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Review*, 51(1):129–159, 2009.
- [12] J. Demmel, J. Dongarra, A. Fox, S. Williams, V. Volkov, and K. Yelick. Accelerating time-to-solution for computational science and engineering. *SciDAC Review*, (15), 2009.
- [13] J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK benchmark: Past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003.
- [14] N. I. M. Gould, D. Orban, and P. L. Toint. CUTER and SifDec: A constrained and unconstrained testing environment, revisited. *ACM Transactions on Mathematical Software*, 29(4):373–394, 2003.
- [15] A. Hartono, M. M. Baskaran, C. Bastoul, A. Cohen, S. K. namoorth, B. Norris, J. Ramanujam, and P. Sadayappan. PrimeTile: A parametric multi-level tiler for imperfect loop nests. In *Proceedings of the 23rd International Conference on Supercomputing, June 8-12, 2009, IBM T. J. Watson Research Center, Yorktown Heights, NY, USA*, 2009.
- [16] A. Hartono, B. Norris, and P. Sadayappan. Annotation-based empirical performance tuning using Orio. In *Proc. of the 23rd IEEE International Parallel & Distributed Processing Symposium*, Italy, 2009.
- [17] A. Kaiser, S. Williams, K. Madduri, K. Ibrahim, D. Bailey, J. Demmel, and E. Strohmaier. TORCH computational reference kernels: A testbed for computer science research. Technical Report UCB/EECS-2010-144, EECS Department, University of California, Berkeley, December 2010.
- [18] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, 2010.
- [19] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Implicit and explicit optimizations for stencil computations. In *Proc. of the ACM SIGPLAN Workshop on Memory System Performance and Correctness (MSPc)*, June 2006.
- [20] T. Kisuki, P. M. W. Knijnenburg, and 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.
- [21] J. D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. <http://www.cs.virginia.edu/stream/>.
- [22] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *IISWC '08: Proceedings of the IEEE International Symposium on Workload Characterization*, pages 35–46, Seattle, WA, USA, 2008.
- [23] J. J. Moré, B. S. Garbow, and K. E. Hillstom. Testing unconstrained optimization software. *ACM Transactions on Mathematical Software*, 7(1):17–41, 1981.
- [24] J. J. Moré and S. M. Wild. Benchmarking derivative-free optimization algorithms. *SIAM Journal on Optimization*, 20(1):172–191, 2009.
- [25] B. Norris, A. Hartono, and W. Gropp. *Annotations for Productivity and Performance Portability*, pages 443–461. Computational Science. Chapman & Hall CRC Press, Taylor and Francis Group, 2007.
- [26] L.-N. Pouchet. PolyBench: The Polyhedral Benchmark suite, 2011.
- [27] A. Qasem, K. Kennedy, and J. Mellor-Crummey. Automatic tuning of whole applications using direct search and a performance-based transformation system. *The Journal of Supercomputing*, 36(2):183–196, May 2006.
- [28] K. Seymour, H. You, and J. Dongarra. A comparison of search heuristics for empirical code optimization. In *Proc. of the 2008 IEEE International Conference on Cluster Computing*, pages 421–429, 2008.
- [29] J. Shin, M. W. Hall, J. Chame, C. Chen, and 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.
- [30] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. *SIGARCH Comput. Archit. News*, 20:5–44, March 1992.
- [31] The pChase Memory Benchmark Page. <http://pchase.org/>.
- [32] A. Tiwari, C. Chen, C. Jacqueline, M. Hall, and J. K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *Proc. of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12, Washington, DC, 2009.
- [33] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178–194, 2009.

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.