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-P1872-0411

April 2011

[1]Mathematics 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

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 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 the 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 finding high-performing parameter configurations quickly 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 developments and experimental studies. In this paper, we describe a set of extensible and portable **s**earch **p**roblems in **a**utomatic **p**erformance **t**uning (SPAPT) whose goal is to aid in the development and improvement of search strategies and performance-improving transformations. SPAPT contains representative implementations from a number of lower-level, serial performance tuning tasks in scientific applications. We present an illustrative experimental study on a number of problems from the test suite. We discuss some 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 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 not portable. Automatic performance tuning or *autotuning* 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 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 [1, 2, 3, 4, 5, 6].

[1]Mathematics 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 our recent work [7], we 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., [8]).

On the other hand, a rich history in mathematical optimization of sets of benchmark problems exists. Examples include the Moré-Garbow-Hillstrom problems for unconstrained optimization [9]; the more general CUTEr set [10] (a subset of which was used as the inputs in [8]); and the smooth, noisy, and nonsmooth problems in [11]. 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 only including problems 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 benchmark set of extensible and portable **s**earch **p**roblems in **a**utomatic **p**erformance **t**uning (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 [12, 13], a recently developed performance-tuning software framework. By making Orio explicitly part of the set and defining specific optimization 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 developments and experimental studies on search algorithms in autotuning.

The rest of the paper is organized as follows. In Section 2 we review related works on benchmark sets 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. [14], Qasem et al. [15], Seymour et al. [16], Shin et al. [17], and Tiwari et al. [5] used a number of linear algebra kernels for autotuning. Pouchet [18] 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. [13] and Hartono et al. [12] used a collection of linear algebra kernels, solvers, and stencils. These are parameterized codes that were used to test the effectiveness of Orio, an annotation-based autotuning framework. 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. [19] 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.

We note that Kaiser et al. [19] argue that a number of existing benchmarks can be seen as reference implementations of one or more kernels from TORCH. Examples include EEMBC [20], HPC Challenge [21], ParBoil [22], SPEC [23], NAS Parallel benchmarks [24], PARSEC [25], Rodinia [26], LINPACK [27], STREAM [28], STAMP [29], SPLASH [30], and pChase [31]. Although in principle these benchmarks can be parameterized and used for autotuning, none of them are developed specifically for evaluating the effectiveness of search algorithms in autotuning. Hence, there is a noticeable void in the literature of benchmark sets of well-formulated search problems in autotuning. The SPAPT set that we propose in this paper is based on [12, 13, 18], which comprises representative examples from autotuning in scientific applications. Moreover, SPAPT has several kernels from [19], and the search problems in SPAPT may be viewed as instances of some TORCH kernels adapted as search problems in autotuning.

Table 1: Collection of benchmark kernels.

Kernel	Operation	Transformations n_i		n_b		$\|\mathcal{D}\|$
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	sub kernel 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

3 Benchmark Set

In this section we provide a high-level overview of the set of benchmarks and the chosen tuning directives. We then discuss their implementations using 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 [32]. 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 [18]: 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 com-

putations 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`); `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 solving implicit and explicit partial differential equations [3]. In this group, we have four kernels from ADI pre conditioners (`ADI`), Jacobi 1-D (`Jacobi-1d`), Seidel stencil (`Seidel`), and 3-D stencils computations (`Stencil3d`).

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 variables, which is central to many statistical packages. The reference implementations are obtained from [18].

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

$$\begin{array}{rll} \min\limits_{x} & f(x) & \\ & x = (x_{\mathcal{B}}, x_{\mathcal{I}}) \in \Omega, & \\ \text{such that} & x_{\mathcal{B}_j} \in \{0,1\}, & j = 1, \ldots, n_b, \\ & x_{\mathcal{I}_j} \in \{l_j, \cdots, u_j\}, & j = 1, \ldots, n_i, \end{array} \tag{1}$$

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]. The feasible set $\mathcal{D}$ for a given problem is defined by bound constraints and a set of more general, algebraic constraints denoted in (1) by Ω. Note that these constraints are typically independent of unsuccessful code evaluations due to transformation, compilation, and run-time errors.

From each tunable kernel, we generate four search problems: three problems by varying the input size (N, $2N$, and $4N$) and the fourth by fixing the value of all binary parameters to 0 (so that only integer decision parameters are considered) with an input size of N. Note that the input size is not limited to single-dimensional or square inputs; for non-square or multi-dimensional inputs, instead of N, we have $\{N_1, N_2, N_3, \ldots\}$.

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). 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 [33, 12].

3.2 Orio-specific implementations

Orio [12, 13] 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 selects 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 [12, 13] 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 `trac.mcs.anl.gov/projects/performance/browser/orio/testsuite/SPAPT.v.01`.

4 Illustrative experiments

In this section, we conduct 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 with 36GB of memory running the stock Linux kernel version 2.6.18 provided by RedHat.

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

prob	$\hat{\mu}_{init}$	$\hat{\sigma}_{init}$
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

4.1 Performance objectives

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 daemon jobs and cold caches. Hence, modeling decisions related to the performance objective can play a significant role in the tuning process.

As a default, we consider minimizing the runtime for each problem. Many performance measures can serve as an optimization objective in (1), including

$$\begin{aligned} f(x) &= \frac{1}{m}\sum_{i=1}^{m} r_i(x), \\ f(x) &= \text{median}_{i=1,\ldots,m} r_i(x), \\ f(x) &= \min_{i=1,\ldots,m} r_i(x), \\ f(x) &= r_3(x), \end{aligned}$$

where $\{r_1(x), \ldots, r_m(x)\}$ denote a sequence of runtime realizations for parameter values x, and these objectives denote the mean, median, minimum, and third realized time, respectively. In SPAPT we intentionally do not specify a fixed form of the objective, because it

can depend heavily on the architecture and the particular performance metric (e.g., runtime, FLOPS, or power). Next we discuss various considerations related to performance objectives given $m = 35$ consecutive replications.

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 ($\hat{\sigma}_{init}/\sqrt{35}\hat{\mu}_{init}$).

Performance objectives other than the mean, including those given above and quantile-based measures, can be adopted based on the ultimate goals of the performance tuning process. Figure 2 shows a comparison of mean, median, minimum, and third runtime values of $5,000$ random parameter configurations x in $|\mathcal{D}|$. Note that all x are sorted with respect to mean. In a problem based on the `ATAX` kernel, we observe that median, min, and third runtime are close to each other. The reason is that they are not sensitive to outliers and cold cache effects. Hence, they can be used as an alternative to the mean when one knows that outliers and/or cold cache effects are rare phenomena in the production runs. However, we have a few exceptions to this general trend in problems such as those based on the `DGEMV` kernels (see Figure 3, where all the performance objectives are similar to each other).

In a number of problems, we found that the median of 35 runs was systematically lower because of the cold cache effect. This is shown in Figure 4, where the execution times of the first few runs are always longer than that of the other runs. Note that the performance objective of third runtime value is explicitly designed to take this into account. In the rest of this section, we use mean runtime as the performance objective.

4.2 Performance objective density

In Figure 5, 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 problems based on the `DGEMV` and `GESUMMV` kernels the number of high-performing parameter configurations is low compared with that for the `BiCG` and `COR` kernels. We expect that a simple random search can find high-performing configurations for problems based on `BiCG` and `COR`, for which there are many high-performing parameter configurations, whereas problems based on the `DGEMV` and `GESUMMV` kernels might require sophisticated search algorithms. Given the large search space, these results should be treated with caution because of the small number of random configurations considered for the experiments. These are indicative results and should not taken as a direct measure for assessing the difficulty of solving a search problem in the benchmark.

4.3 Parameter formulations

A possible approach to solving search problems in autotuning is to treat integer parameters as real valued ones and to use advanced numerical optimization algorithms. However, the

presence of binary parameters makes this approach less promising. Hence, from a numerical optimization standpoint, it is interesting to analyze distribution of the objective values when binary parameters are set to some default values. In Figure 6, we compare the results of first and fourth problems based on the `BiCG` kernel. Recall that a fourth problem for each kernel is obtained from the first problem by setting the binary parameters to 0. The results show that switching off the binary parameters leads to high-performing configurations: the range 0.002 to 0.007 of Figure 6(b) belong to the first bin of the histogram in Figure 6(a), which constitutes only 5%. However, the best parameter configuration with an objective value of 0.00012 is not feasible in the fourth problem because it has two binary parameters set to 1.

4.4 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 situation is illustrated in Figure 7, which shows the correlation between the objectives for different instance sizes. In problems based on the `ATAX` kernel (see Figure 7(a)), a large number of high-performing parameter configurations for input size N becomes less effective for input size $4N$. This 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 in cache. Nevertheless, the results from problems based on the `BiCG` kernel (see Figure 7(b)) show that high-performing parameter configurations are generalizable to some extent for certain types of computations.

5 Conclusions and Future Directions

Motivated by a lack of benchmark set of search problems in autotuning, we developed SPAPT, a collection of representative kernels from scientific applications that are good candidates for autotuning. Each search problem comprises parameterized tuning directives, values for each parameter, input sizes, and an initial configuration for search 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 some illustrative experiments to show performance impacts of problem characteristics such as choice of performance objectives, noise, cold cache effects, binary parameters, and input sizes.

SPAPT has the potential to improve the state of the art in autotuning. On the one hand, it can help the autotuning community conduct systematic experimental studies, which will help the development and evaluation of transformation approaches. On the other hand, our easily accessible, portable Orio implementation of the benchmark suite can attract

mathematical optimization researchers to develop search algorithms without knowing the fine details of compiler optimization and performance tuning.

In addition to the limitations of any of the benchmarks described in Section 1, SPAPT has the following ones. The benchmark deals only with serial problems and does not provide any parallel problems. As a starting point, we focused on some of the widely used scientific kernels in the autotuning literature. A possible bias in the benchmark set is that a large number of problems deal with linear algebra related computations.

We plan to continue to extend the application space and numerical and scientific problem domain coverage of the benchmark set. In particular, we will define search problems using additional kernels from TORCH. We will use SPAPT to understand the search problem characteristics, to benchmark the existing search algorithms for autotuning, and to develop new search techniques. We are also planning to analyze the impact of different architectures on the SPAPT problems. In future, we also intend to build a database of tabulated execution times to facilitate benchmarking search algorithms.

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.

References

[1] I. Chung, 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.

[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. Kamil, C. Chan, L. Oliker, J. Shalf, S. Williams, An auto-tuning framework for parallel multicore stencil computations, in: Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on, 2010, pp. 1–12.

[4] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, K. Yelick, Implicit and explicit optimizations for stencil computations, in: Proc. of the ACM SIGPLAN Workshop on Memory System Performance and Correctness (MSPc), 2006.

[5] A. Tiwari, C. Chen, C. Jacqueline, M. Hall, J. K. Hollingsworth, A scalable auto-tuning framework for compiler optimization, in: Proc. of the 2009 IEEE International Symposium on Parallel & Distributed Processing, Washington, DC, 2009, pp. 1–12.

[6] 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.

[7] P. Balaprakash, S. Wild, P. Hovland, Can search algorithms save large-scale automatic performance tuning?, in: The International Conference on Computational Science, 2011.

[8] 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.

[9] J. J. Moré, B. S. Garbow, K. E. Hillstrom, Testing unconstrained optimization software, ACM Transactions on Mathematical Software 7 (1) (1981) 17–41. doi:http://doi.acm.org/10.1145/355934.355936.

[10] 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.

[11] J. J. Moré, S. M. Wild, Benchmarking derivative-free optimization algorithms, SIAM Journal on Optimization 20 (1) (2009) 172–191.

[12] A. Hartono, B. Norris, P. Sadayappan, Annotation-based empirical performance tuning using Orio, in: Proc. of the 23rd IEEE International Parallel & Distributed Processing Symposium, Italy, 2009.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] L.-N. Pouchet, PolyBench: The Polyhedral Benchmark suite (2011).
URL `http://www-roc.inria.fr/~pouchet/software/polybench/`

[19] 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`

[20] The embedded microprocessor benchmark consortium, http://www.eembc.org.
URL `http://www.eembc.org`

[21] HPC Challenge benchmark, http://icl.cs.utk.edu/ hpcc/.

[22] The Parboil benchmark suite, http://impact.crhc.illinois.edu/parboil.php.
URL `http://impact.crhc.illinois.edu/parboil.php`

[23] SPEC benchmarks, http://www.spec.org/ benchmarks.html.
URL `http://www.spec.org/benchmarks.html`

[24] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, S. Weeratunga, The NAS Parallel Benchmarks, International Journal of High Performance Computing Applications 5 (3) (1991) 63–73.

[25] 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

[26] 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.

[27] J. Dongarra, P. Luszczek, A. Petitet, The LINPACK benchmark: Past, present and future, Concurrency and Computation: Practice and Experience 15 (9) (2003) 803–820.

[28] J. D. McCalpin, Stream: Sustainable memory bandwidth in high performance computers, Tech. rep., University of Virginia, Charlottesville, Virginia, a continually updated technical report. http://www.cs.virginia.edu/stream/ (1991-2007).
URL http://www.cs.virginia.edu/stream/

[29] C. C. Minh, J. Chung, C. Kozyrakis, K. Olukotun, STAMP: Stanford Transactional Applications for Multi-Processing, in: IISWC '08: Proceedings of the IEEE International Symposium on Workload Characterization, Seattle, WA, USA, 2008, pp. 35–46.

[30] J. P. Singh, W.-D. Weber, A. Gupta, SPLASH: Stanford parallel applications for shared-memory, SIGARCH Comput. Archit. News 20 (1992) 5–44.

[31] The pChase Memory Benchmark Page, http://pchase.org/.

[32] J. Demmel, J. Dongarra, A. Fox, S. Williams, V. Volkov, K. Yelick, Accelerating time-to-solution for computational science and engineering, SciDAC Review (15).

[33] A. Hartono, M. M. Baskaran, C. Bastoul, A. Cohen, S. K. namoorth, B. Norris, J. Ramanujam, 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. W atson Research Center, Yorktown Heights, NY, USA, 2009.

```
  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 ACOPY_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;
  }
/*@ end @*/
```


Figure 1: Parameter specification and constraint example in Orio.

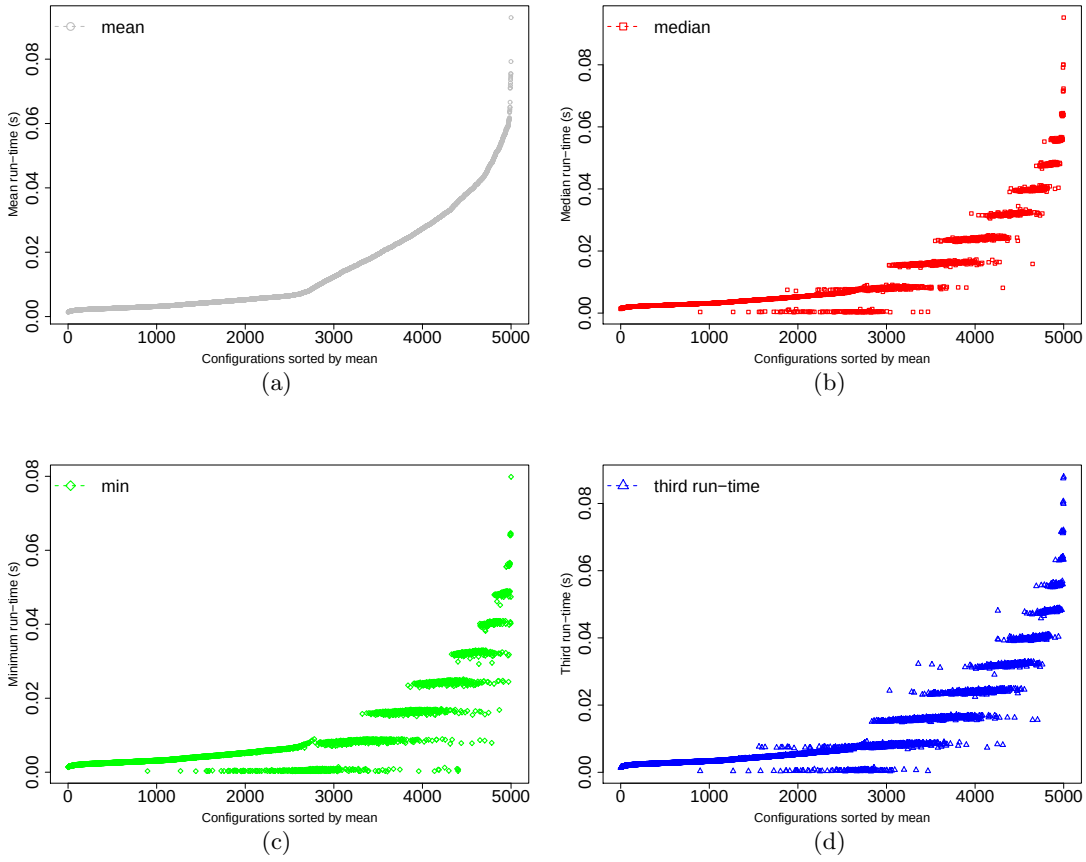


Figure 2: Comparison of performance objectives in a SPAPT problem based on the ATAX kernel.

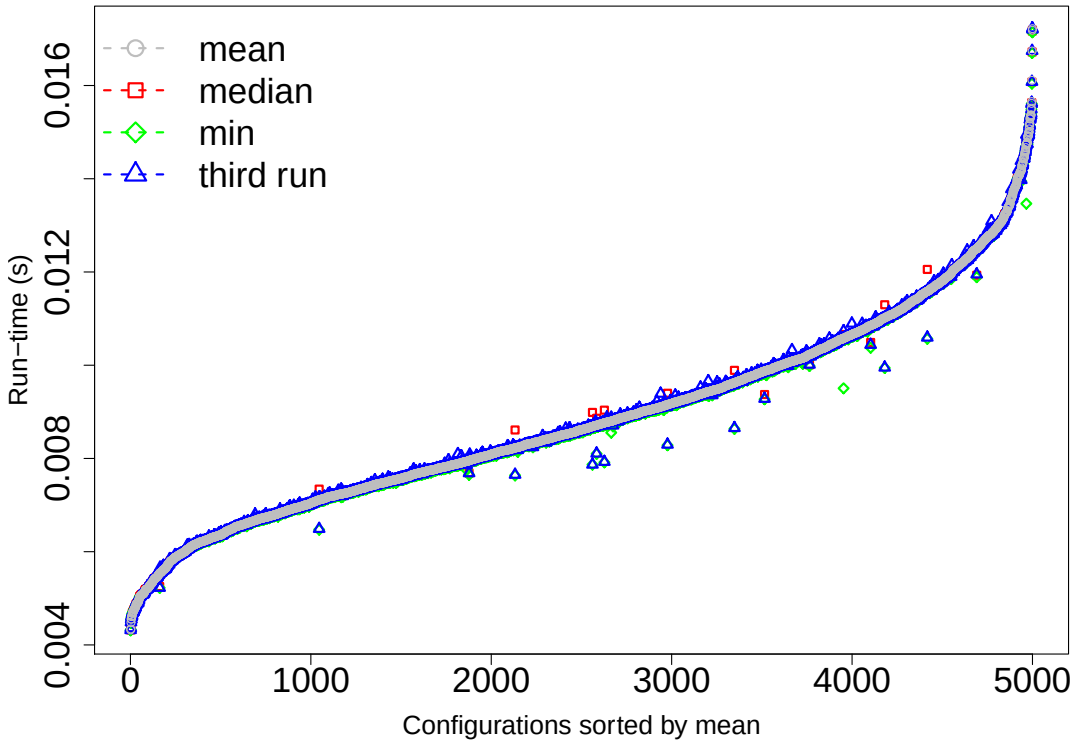


Figure 3: Comparison of performance objectives in a SPAPT problem based on the DGEMV kernel.

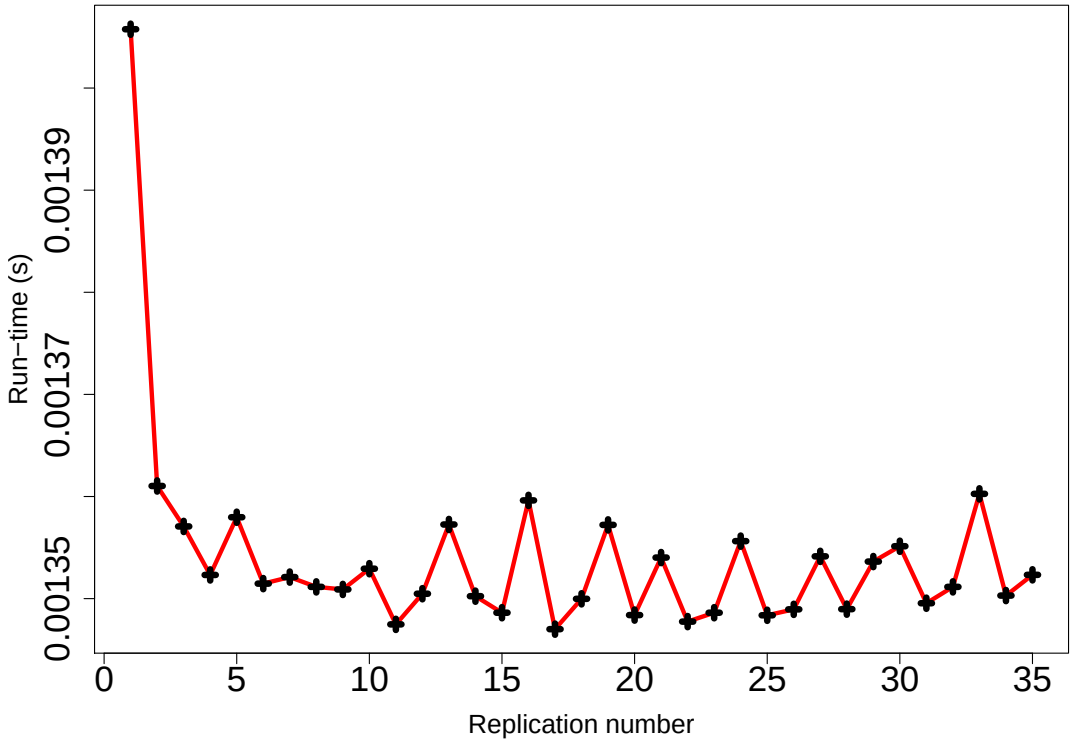
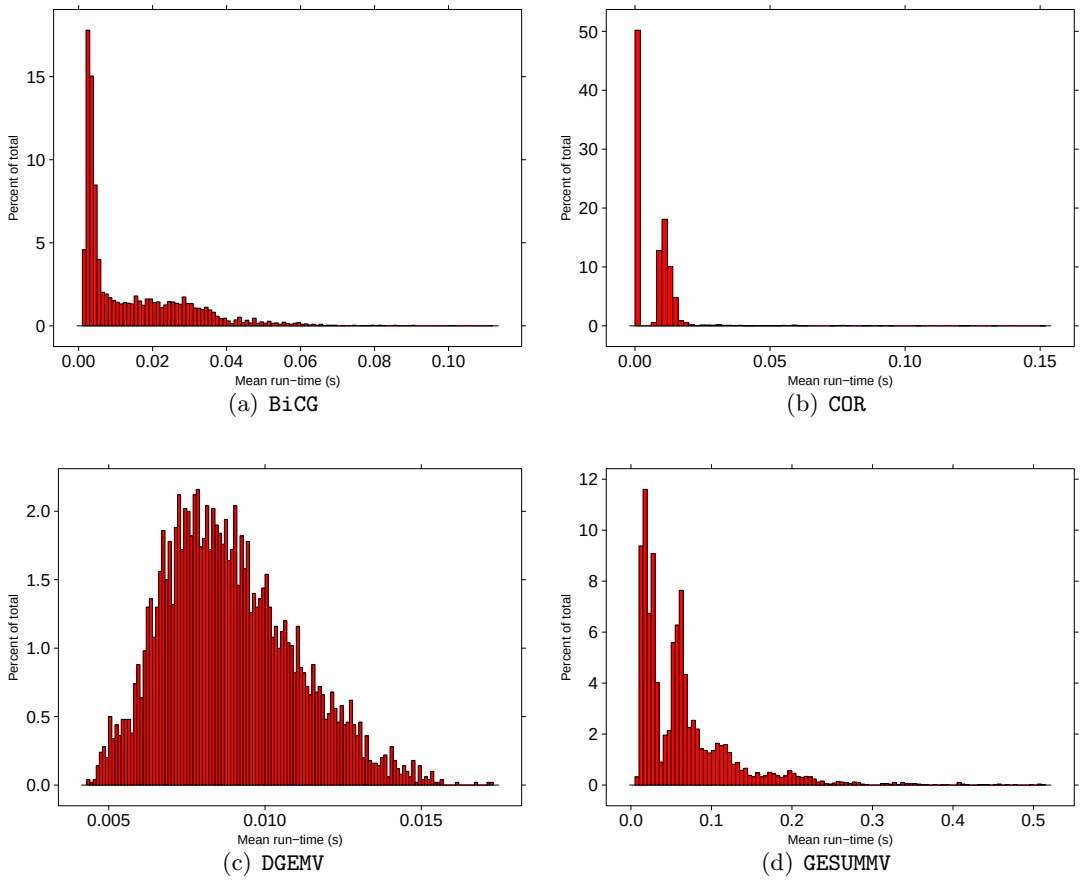


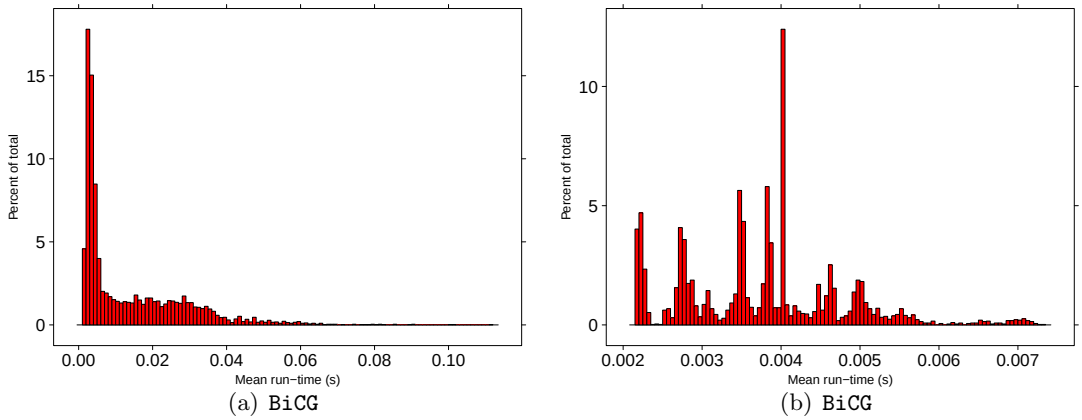
Figure 4: The cold cache effect in a SPAPT problem based on the `ATAX` kernel: run-time realization as a function of replication number.



(a) `BiCG` (b) `COR`

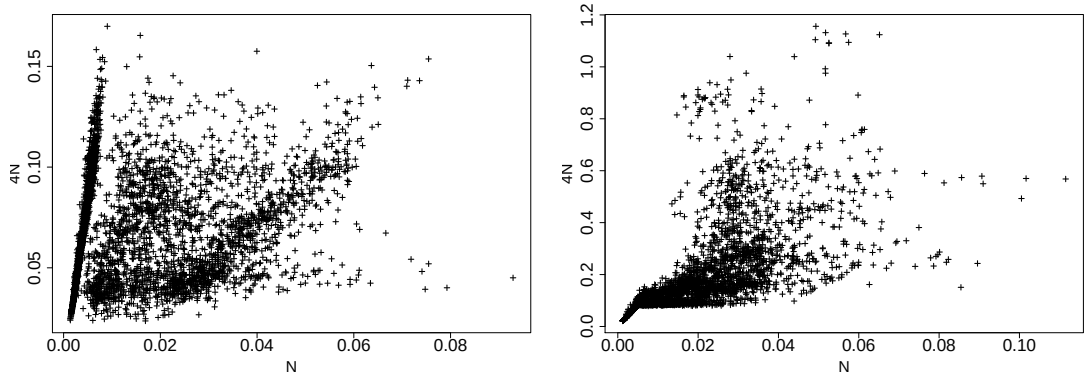
(c) `DGEMV` (d) `GESUMMV`

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



(a) `BiCG` (b) `BiCG`

Figure 6: Impact of binary parameters: histograms of objective values from 5,000 random code variants in $\mathcal{D}$.



(a) `ATAX` (b) `BiCG`

Figure 7: Impact of input size.