Karlsruher Institut für Technologie

Karlsruhe Reports in Informatics 2011,29
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

Evolutionary Auto-Tuning for Multicore Applications

Andreas Zwinkau and Victor Pankratius

2011

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

Evolutionary Auto-Tuning for Multicore Applications

Andreas Zwinkau
Karlsruhe Institute of Technology
76131 Karlsruhe, Germany
zwinkau@kit.edu

Victor Pankratius
Karlsruhe Institute of Technology
76131 Karlsruhe, Germany
www.victorpankratius.com
pankratius@ipd.uka.de

ABSTRACT

Multicore processors have conquered desktops PCs, servers, and embedded platforms. Parallel computing is now available for a large spectrum of applications, many of which are non-numerical. The increasing parallel hardware and software diversity, however, poses great challenges for programmers. They struggle with application performance optimization and have to account for many diverse and interdependent software parameters that need a proper configuration on every platform. But even if applications are adaptive, a naive approach of manually configuring application performance parameters on each platform becomes infeasible. Large search spaces and long run-times make exhaustive searches impractical, and intuitive values can miss sweet spots altogether. We tackle this important problem and present a domain-independent automatic performance tuning approach that works with a large spectrum of applications. We introduce a novel infrastructure in Eclipse that completely automates the application tuning process using evolutionary search strategies, and which is easy to use by average programmers. Our approach improves portability and works for numerical and non-numerical programs. To tune a program, a feedback-directed optimizer collects run-time information to predict parameter configurations that are likely to lead to good performance in future runs. Our technique generalizes auto-tuning to make it applicable for a variety of application-level parameters and programs in different domains. We quantify the effectiveness of various tuning strategies on a diverse set of applications and multicore platforms. We provide comparative evidence that evolutionary strategies optimize well, most notably significantly better than the simplex-based search algorithms that are predominantly used in the literature. Our insights are grounded on evidence thoroughly gathered from model-based analyses as well as from performance analyses with real programs, including non-numerical programs.

Technical Report 2011-29
Karlsruhe Institute of Technology (KIT), Germany
Institute for Program Structures and Data Organization (IPD)
October 4, 2011

1. INTRODUCTION

Multicore processors with several cores on a chip are mainstream, and programmers are now challenged to parallelize all kinds of performance-critical applications. A problem that makes multicore programming hard is that multicore platforms differ, e.g., in the number of supported hardware threads, memory size, memory bandwidth, cache size, cache architecture, libraries, and operating systems. Consequently, software optimized for one platform might not perform well on other platforms, which harms portability. This situation requires that multicore software becomes adaptive to exploit all available performance potential to the limit.

In the past, auto-tuning strategies have been proposed to perform such adaptations, however, they predominantly focus on numerical programs in particular domains, such as FFT, signal processing, and matrix multiply [8, 9, 14, 24]. The approaches are so specific that they will not fit to applications that do not perform any of these tasks, which is the case for many multicore applications on desktops and servers. Moreover, relying on compilers to do performance tuning typically does not exploit the parallelism potential to the limit, as compiler optimizations can be too fine-granular and too low-level [1, 18, 23], thus missing important leverage.

To compensate these shortcomings and introduce more adaptivity, software engineers nowadays parameterize their parallel programs on an application level. Such parameters can control program tuning knobs that affect performance on various granularity levels beyond loop unrolling. For example, parameters are commonly used to set the number of threads in application thread pools, configure various buffer sizes, maximum number of workers, size of data partitions, choices of algorithms and parallel patterns, etc. However, programmers now face a new problem: All performance parameters p_i can span a potentially large configuration search space for one program. For k parameters, the entire search space consists of the cross product of all domains, i.e., $dom(p_1) \times dom(p_2) \times \ldots \times dom(p_k)$. In general, exhaustive search is unrealistic due to the size and program run-times that are required to map the space. Guessing can be ineffective, and overgeneralized intuitive heuristics, such as assuming that the number of software threads must equal the number of cores, can easily miss performance sweet spots: for example, applications where more threads hide latency would miss speedup opportunities, whereas applications where more threads increase synchronization overhead would slow down.

These problems are highly relevant in practice, which is why we tackle them in this report with a general approach. In particular, this report makes the following novel contributions. We introduce a domain-independent automated approach to tune on various platforms a large spectrum of different multicore applications, such as video encoding, image processing, ray tracing, clustering, data mining, simulations,

content search, compression, and others. Finding tuning approaches that work well in a breadth of domains, rather than in one single domain, is a difficult problem. We present a novel extensible tuning infrastructure for the Eclipse development environment. This infrastructure relieves software engineers from trial-and-error application tuning and is easy to use during everyday development and porting of programs. Our technique employs a feedback-directed dynamic optimizer that can be attached to various applications. Motivated by the promising results that evolutionary strategies have shown on difficult optimization problems [13, 15, 19], we introduce extensions and show that evolutionary search is a good match for the breadth of workloads in our performance tuning context. One of the reasons for this match is the way these algorithms combine systematic search and randomized search. Moreover, we provide a thorough evaluation and compare the effectiveness of other popular techniques such as simplex-based tuning, swarm tuning, and random tuning, including variants thereof. The report provides evidence that evolutionary search yields the best tuning results and outperforms simplex approaches that are most commonly used in literature (e.g., in [20]). Our insights are grounded on both model-based analyses as well as performance analyses with real programs from the PARSEC [2] benchmark suite. We emphasize that in contrast to prior work [8, 9, 14, 24] our approach does not require a particular type of program and also works for non-numerical programs. We concentrate on an application level to leverage additional performance, as opposed to lower-level compiler optimizations [1, 18, 23], but we do not exclude that other forms of auto-tuning can be used in combination.

This report is organized as follows. Section 2 details the optimization problem. Section 3 introduces our novel multicore performance tuning framework. Section 4 presents our evolutionary tuning strategies. Sections 5 and 6 describe adaptations of particle swarm tuning, and simplex- and polytope-based methods as commonly used approaches. Section 7 evaluates and compares all tuning techniques. Section 8 contrasts related work. Section 9 provides our conclusion.

2. THE PROBLEM

Our particular optimization problem is related to offline tuning and can be formulated as follows. Given a multicore program P that has k performance-relevant parameters (assumed to be accessible via command line), the goal is to minimize P's execution time in an iterative way. The optimization process starts with an initial parameter value configuration, executes P, measures run-time, and calculates a new parameter configuration. The process repeats until some termination condition holds, e.g., reaching a given number of executions or a performance improvement below a certain threshold. Tuning is carried out prior to production use of the program. During production use, the program uses the best parameter configuration.

We model a *program configuration* with k parameters as a multi-dimensional vector $x \in \mathbb{N}^k$. Let the run-time measurement of a program with parameters k be a function $t : \mathbb{N}^k \rightarrow \mathbb{R}$. Our performance optimization problem can then be formulated as a multi-dimensional minimization problem: $\text{argmin}(t(x))$ for $x \in \mathbb{N}^k$. Since program parameter configurations are elements of the vector space $\mathbb{N}^k$, the minimization of t is equivalent to the search of the smallest element in $\mathbb{N}^k$, where the comparison of two configurations x and $y \in \mathbb{N}^k$ is determined by their corresponding run-time: $x < y \Leftrightarrow t(x) < t(y)$. The function t is discontinuous and non-differentiable, so analytical methods like gradient decent or Newton's method are not applicable.

Our approach is based on empirical search methods [11] that use just function evaluations during optimization and that don't require derivatives. A problem for analysis, however, is that the evolutionary search methods use randomization, which makes their comparison in a purely analytical way difficult. We are also targeting a breadth of application domains, which would require a multiple of the modeling effort. Thus, we compare results using representative empirical metrics.

2.1 Comparison Metrics

We employ two common empirical metrics to characterize and compare tuning strategies. A first metric is the *tuning error*, i.e., the distance between the optimum found by an algorithm and the real optimum (known in our benchmarks). The second metric is the *number of program executions* of P (which in our designs is also the number of tuning algorithm iterations). This is motivated by the fact that run-times can vary for different programs, from seconds to days. Parameter reconfiguration occurs between program runs, so tuning time and tuning overhead does not affect program run-time. Moreover, tuning overhead is typically dwarfed by P's execution time. The number of program evaluations has therefore higher practical relevance compared to fixing the maximum time available for tuning. Nevertheless, as tuning overhead is negligibly small compared to one program execution, the aggregated number of tuning iterations in offline tuning runs multiplied by the program execution time can be interpreted as an estimate for the total time spent on tuning.

To compare tuning strategies, however, it would be misleading to just look at the sum of program execution times during offline tuning plus some overhead between runs. This would show a distorted picture in our scenario with different domains, because program run-times can differ significantly. Programs may with short run-times have a chance to perform more evaluations with certain algorithms and thus end up better because the optimization has advanced more, whereas programs with long run-times would be at a disadvantage. For these reasons, we consider that the number of evaluations is a more objective metric for a cross-comparison of tuning strategies.

3. A SOFTWARE FRAMEWORK FOR MULTICORE APPLICATION PERFORMANCE TUNING

As a proof of concept, we present a fully-functional multicore performance tuning infrastructure. We employ this infrastructure for the evaluations presented later in this report. The tool supports developers in the process of tuning multithreaded applications that expose performance-relevant parameters on the command line.

Figure 1 shows the user interface, which is part of the Eclipse development environment to support software engineers and relieve them from manual tuning tasks. The figure shows one of the configuration screens to connect the tuner to an application and let it know about the tuning parameters. The other screenshots show a summary of the multidimensional search space sampled by our auto-tuner.

Our tuner executes a program in an iterative way and gathers run-time feedback as described in Section 2. Program run-time is optimized using multi-dimensional algorithms and the parameter constraints defined by the developer. All algo-

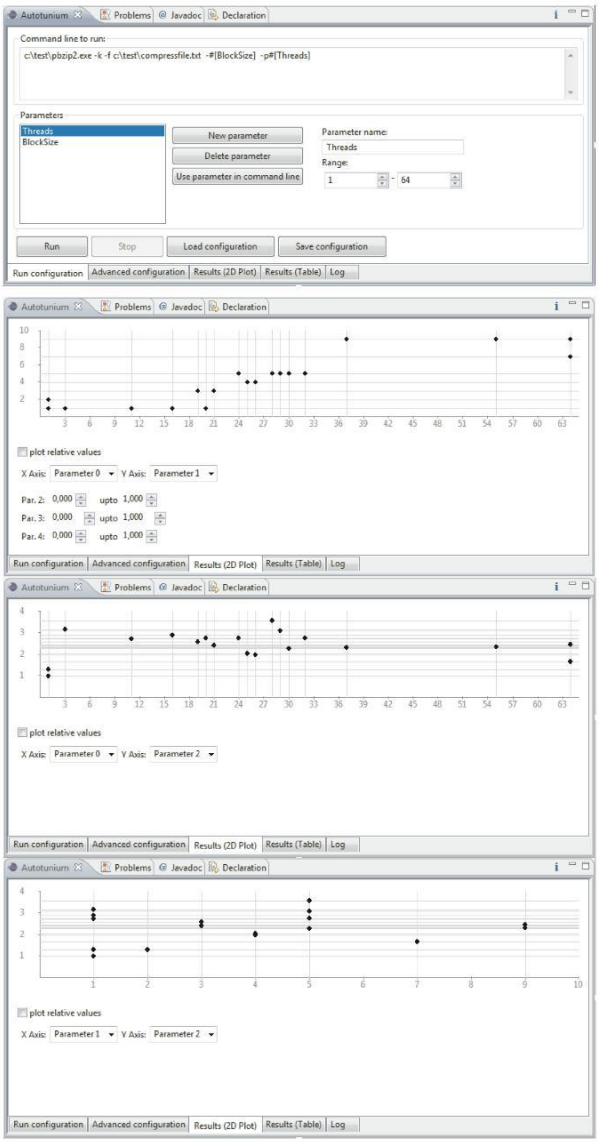


Figure 1: Screenshot of our tuning graphical interface in the Eclipse development environment [12].

rithms and numerical methods are carefully ensured to work on a discrete space. It is also possible to define optimization goals other than run-time and express minimization and maximization problems.

Our tuner is implemented in Java, but it can connect to and tune all sorts of binary applications that are written in other languages. A particular feature of our infrastructure is that it is extensible: All tuning algorithms can be implemented in a plug-in style in Java. Plug-ins do not require recompilation and can be treated as scripts. It is possible to offer future plug-ins in Web repositories, so programmers can easily update the search strategies used by their tuners. To keep results comparable, all algorithms presented in this report are implemented as part of this infrastructure.

3.1 Extending the Infrastructure with User-Defined Tuning Strategies

As an example on how to extend the tuning infrastructure, we briefly describe the mechanisms and interfaces.

A new tuning algorithm inherits from an abstract super class `OptimizationAlgorithm`, as illustrated in Figure 2 [1].

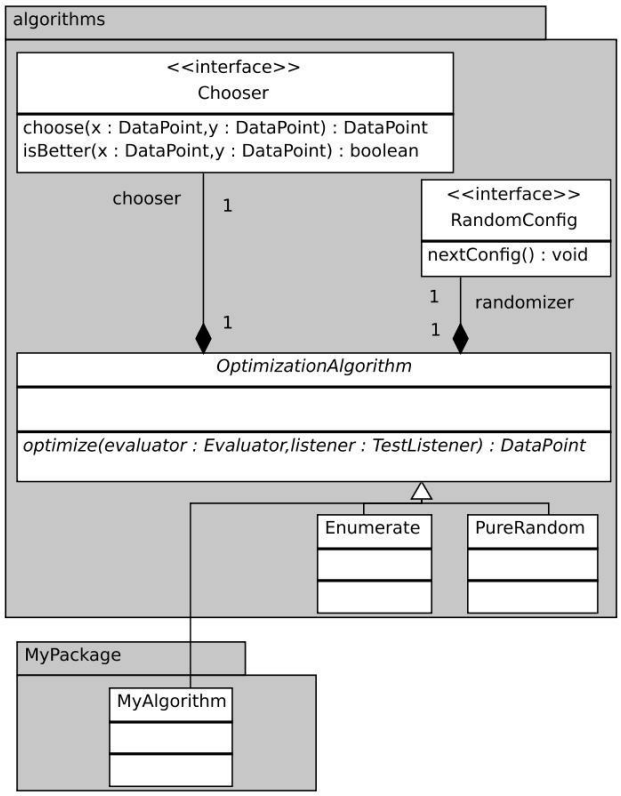


Figure 2: Class diagram for optimization algorithms

A tuning strategy developer implements the abstract method `optimize`; this method implements the tuning algorithm and returns the result as a `DataPoint` object, which contains the optimal configuration and the associated program execution time. An `Evaluator` object abstracts from the program to execute; the rationale is that execution and measurement are tasks that should be separated from the optimization algorithm. This abstraction allows using different evaluators, for example to gather the execution time of a program or just read the execution time from a recorded database. The `TestListener` object monitors the optimization process; for example, it calls the graphical user interface to update displays. The tuning algorithm programmer must take care to call the listener in the appropriate places.

A tuning algorithm uses the `chooser` attribute to avoid hard-coding whether to search for minima or maxima. The `choose` method selects one of two `DataPoint` objects, and the developer defines which one is the "better" one.

For example, the "Enumerate" plug-in implements a tuning algorithm that enumerates all parameter value configurations, one after the other:

[1]The UML diagram is simplified and shows just essential parts. For example, the chooser and randomizer attributes are private and must be accessed by getter/setter methods.

Listing 1: Enumerative Search: Java code

```
public class Enumerate
      extends OptimizationAlgorithm {
  @Override
  public DataPoint optimize(
        Evaluator e,
        TestListener l) {
    DataPoint best = null;
    do {
      l.beforeTest(best);
      DataPoint test = e.eval();
      l.afterTest(test);

      best = getChooser()
          .choose(test, best);
    } while (nextConfig());
    return best;
  }
}
```

`Enumerate` extends the abstract `OptimizationAlgorithm` class of the auto-tuner and overrides the `optimize` method. The `optimize` method takes an evaluator – which implements the optimization function t mentioned earlier – and a listener object, which is used for feedback to the GUI.

The enumeration of all configurations is implemented with a do-while-loop that moves from the current configuration to the next by calling `nextConfig()`, until no further configuration is available. Inside the loop, the actual evaluation calls the `e.eval()` method. The last line in the loop selects the currently best `DataPoint` object by comparing the best value so far with the new measurement. After finishing with all configurations, the algorithm returns the best configuration.

4. DOMAIN-INDEPENDENT TUNING BASED ON EVOLUTIONARY SEARCH

Evolutionary optimization has been shown to work generally well on complex optimization problems [13, 15, 19]. This motivates us to develop evolutionary approaches for multicore performance optimization.

4.1 Basic Evolution

Evolutionary algorithms operate on a population of individuals. New individuals (i.e., in our case, performance configurations) evolve using mutation and selection operators [15]. Each individual has a fitness value, which in our case is the associated execution time. Algorithm 1 sketches performance tuning with a population of size k.

Algorithm 1 Basic Evolution Tuning

$p \in (\mathbb{N}^n)^k$, a set of configuration vectors
for g generations **do**
  $p \leftarrow \text{selection}_k(p \cup \{\text{MUTATION}(p)\})$
return $\text{selection}_1(p)$

procedure MUTATION(p)
  b = best vector of p
  s = second best vector of p
  r = random vector from p
  return $\alpha b + \beta s + \gamma r$

In each generation, we create one new individual by mutation. The $selection_k$ operator selects the k best individuals for the next generation by mixing the two best individuals with a random individual. This step keeps individuals with good performance characteristics.

We generate new mutants by mixing the two best individuals and one random individual. In addition, we employ randomization to potentially escape local minima. The influence of each individual is determined by a weight $\alpha, \beta, \gamma \in \mathbb{R}$, where $\alpha + \beta + \gamma = 1$ and $\alpha = 0.3, \beta = 0.5, \gamma = 0.2$. These parameters focus the search in the area around the best individual but also provide enough weight to escape potential local minima, which is what we need in multicore application performance tuning.

Finding a suitable termination condition requires a compromise. If individuals flock at two different local optima, the algorithm might not terminate if the condition requires a vicinity of ϵ. Stopping after certain decreases in improvement might miss important parts of the search space. Our explorative studies have shown that the following approach is effective: We limit the search to a number of generations logarithmic in relation to the search space and stop after $g = d \cdot \log(1000n)$ generations, where d is the dimension of the search space and n the number of configurations.

4.2 Differential Evolution

As shown in Algorithm 2, Differential Evolution uses a mutation method that differs from Basic Evolution.

Algorithm 2 Differential Evolution Tuning

procedure MUTATION(p)
  select $p_1, p_2, p_3, p_4 \in p$ randomly
  return $\text{MIX}_\alpha(p_1 + F \cdot (p_2 - p_3), p_4)$

procedure $\text{MIX}_\alpha(x, y)$
  for all $i \in \{1, \ldots, |x|\}$ **do**

$$z_i = \begin{cases} x_i & \text{with probability } \alpha \\ y_i & \text{else} \end{cases}$$

  return z

The mutation operator [19] picks four random vectors $p_1, \ldots, p_4$. It scales the difference $p_2 - p_3$ two by a differential weight factor $F \in]0, 2]$ and calculates $v = p_1 + F \cdot (p_2 - p_3)$. To increase diversity, p_4 is mixed with v to produce a new vector $n = \text{mix}_\alpha(v, p_4)$. Elements from p_4 are selected with probability α and from v with probability $1 - \alpha$, where $F = 0.3$ and $\alpha = 0.2$. The resulting vector becomes part of population p, and the individual with the worst fitness value is removed.

4.3 Balanced Evolution

We introduce balanced evolution as a new variant of evolutionary tuning. Starting configurations are initialized by the boundary points in the search space, i.e., the points where the values of each dimension are minimal or maximal. For example, a 2-dimensional search space $[1, n] \times [1, m]$ has the boundary $\{(1, 1), (n, 1), (1, m), (n, m)\}$. In addition, random configurations are added to the initial population until it has the same size as the initial populations of the other evolutionary algorithms.

To evaluate a new point in the search space, the process is guided as follows. Firstly, we aim to reduce uncertainty and avoid that some parts of the search space are not covered at all. Secondly, we explore configurations that have most potential to represent the global optimum. We assume that points with good performance are in the neighborhood of other points with good performance found so far.

We define a heuristic assigning an "interestingness" measure $i_p = uncertainty - potential$ to a point $\vec{p}$ in the search

Algorithm 3 Balanced Evolution Tuning

```
p ← starting population
for g generations do
    p ← p ∪ {MUTATION(p)}
return selection_1(p)

procedure MUTATION(p)
    select x with maximum i_x
    return {x}
```

space. In particular, $uncertainty = |\vec{n}_p - \vec{p}|$ and $potential = (t(\vec{n}_p)+t(\vec{n}'_p))/2$, where $\vec{n}_p$ is the nearest evaluated point to $\vec{p}$, and $\vec{n}'_p$ the second nearest. *uncertainty* steers the coverage of the search space, whereas *potential* steers search convergence. When generating a new individual, the algorithm selects the point p with maximum i_p. The algorithm terminates when the maximum number of generations is reached. The selection of the best individual occurrs after termination, as an earlier removal of individuals from the population would lead to loosing the "interestingness" measures so far. In extremely rare cases where $\vec{n}_p$ and $\vec{n}'_p$ are too close to each other and the population does not change, the algorithm is reset.

4.4 Unbalanced Evolution

We create a variant of balanced evolution that ignores the uncertainty of Balanced Evolution. This strategy leads to a faster convergence. It follows the direction of configurations that are most likely optimal and avoids unknown territories. It risks, however, getting stuck in local optima.

5. PARTICLE SWARM TUNING

Particle swarm tuning [10] works in a similar way to evolutionary tuning. In principle, particles float through the search space with a certain inertia and are expected to flock eventually around the global minimum.

Algorithm 4 Particle Swarm Tuning

```
initialize particle swarm S
b ∈ S with t(b) minimal
for k steps do
    for p ∈ S do
        //p* is the best in p's history
        //d_p is p's current movement
        //vector
        d_p ← d_p + α(b − p) + β(p* − p)
        p ← p + d_p
        if t(p) > t(b) then
            b ← p
return b
```

An initial swarm consists of a set S of random points from the search space. Each particle $\vec{p}$ has a movement vector $\vec{d_p}$. Furthermore, let $\vec{p^*}$ be the best configuration of p so far and $\vec{b}$ the best of all $\vec{p^*}$. In iteration i, each particle's movement vector $\vec{d_p}$ is adjusted by $\vec{d_{p,i}} = \vec{d_{p,i-1}}+\alpha(\vec{b}-\vec{p})+\beta(\vec{p^*}-\vec{p})$, with $\alpha = 1.1$ and $\beta = 0.3$. Each particle's new position is then determined by $\vec{p_i} = \vec{p_{i-1}} + \vec{d_{p,i}}$. If in rare cases particles swap over the search space, they are pushed back to the closest feasible location.

Our particle count is logarithmic in relation to the search space size, and step count is linear in dimension size. The algorithm terminates after a predefined step count. This design makes results easier to compare with the algorithms discussed in the other sections.

6. SIMPLEX- AND POLYTOPE-BASED TUNING

Nelder and Mead [4] use a simplex moving through the search space to find minima. A simplex $s \in (\mathbb{R}^d)^{d+1}$ is the simplest polygon for an arbitrary dimension d (e.g., a triangle in two-dimensional space). The popular method illustrated in Algorithm 5 uses three parameters α, β and γ, with $\alpha = 1.1, \beta = 0.65, \gamma = 2.0$. In every step, only the worst node is moved. If a point is moved outside the search space, it is pushed back into valid space with a small random displacement.

Algorithm 5 Simplex-based Tuning

```
s ∈ (ℕ^n)^{n+1}
n ← reflexion_α(s)
while u > |s| · d do
    if t(n) > t(worst(s)) then
        n ← n + β · (best(s) − n)
        if t(n) > t(worst(s)) then
            compress_β(s)
            n ← reflexion_α(s)
    else
        if t(n) < t(best(s)) then
            n ← n + γ · (n − worst(s))
        else
            s ← (s ∪ {n}) \ {worst(s)}
            n ← reflexion_α(s)
return m
```

Our termination condition is based on the distance sum u of every simplex node to the best simplex node. The rationale is that simplex points will get closer together when a minimum is approached. As we operate on discrete values the simplex cannot contract below a certain limit, so it stops when $u \leq |s| \cdot d$.

Polytope-based Tuning. A simplex is a special case of a polytope. The simplex-based tuning algorithm can be generalized to employ a polytope with $s \in (\mathbb{R}^d)^x$ and $x > d + 1$ instead of a simplex, assuming that more points will improve result quality.

For an appropriate value of x, two factors have to be balanced. On the one hand, x should be large, because more information is available to make a decision. On the other hand, x should not be too large to cause a large number of initial evaluations. This would make the approach too expensive as it would require many repeated program executions. Our evaluations reveal that there is no significant difference for $x \in \{2d, 4d, 8d\}$, so our polytope implementation is initialized with the middle one, $x = 4d$. The rules for polytope-based optimization are the same as for simplex-based optimization.

7. COMPARATIVE EVALUATION AND ANALYSIS

Which approach finds the best performance parameter configuration? This question is difficult to answer entirely analytically, for several reasons. The algorithms employ randomization that makes their behavior hard to predict. Also, making too many assumptions to simplify the problem would lead to a significant loss of realism and practical utility of our findings. Therefore, we provide an answer from two angles: (1) using a model-based approach, as well as (2) a set of real parallel programs that are tuned on 4-core and 8-core platforms.

Starting with models has the advantage that we can eliminate system-related noise and analyze tuning behavior in a controlled environment. This allows us to characterize and explain key factors affecting tuning effectiveness. In the next step, we benchmark our approaches with a variety of real-world multicore applications. Our insights are highly valuable for parallel programmers who are under pressure to produce good results quickly.

A particular problem in evaluating and comparing all tuning algorithms is that the number of iterations (i.e., how often they execute a program that is tuned) cannot be controlled in all of them. So it is not possible to keep this parameter constant, try out all algorithms, and select a winner based on the lowest-found program execution time. It happens that one algorithm needs many evaluations and achieves a good result, whereas another algorithm needs fewer iterations for a worse result. One cannot say that "20 tuning iterations leading 10s program run-time" is better than "10 tuning iterations leading to 30s program run-time"; it depends on the preference of the person who tunes a program whether he or she favors fewer iterations or lower run-times. This is why we conduct several analyses from different perspectives to assess these tradeoffs. We employed percentile boostrapping [7] to estimate confidence intervals and make sure that our results are within acceptable ranges. We are particularly interested in the strategies that provide the best runtime.

7.1 Model-Based Search Space Analyses

7.1.1 Performance Models and Search Space Shapes

We start with models of multicore program performance to analyze the tuning strategies in a controlled environment. De Jong [6] collected a set of five functions that are commonly used to stress-test optimization approaches: Sphere, Rosenbrock's Saddle, Steps, Biquadratic Function with noise, and Shekel's Fox Holes function. We included three additional functions to increase variety. Examples for some function shapes in three dimensions are shown in Figure 3; each function models the run-time of a program as the dependent variable and two performance-impacting parameters as the independent variables. In the multidimensional case we employ $f(x_1, \cdots, x_n) = \prod_{i=1}^{n} f(x_i)$, and a "Holes" function created algorithmically according to [6]. All functions are discretized and scaled to $[1, 1000]$. Up to 10% of noise is added to simulate fluctuations of real measurements. The minimum of every function is 1, and no function has a value greater than 1100.

7.1.2 An Aggregated Comparison

For an initial overview of how tuning strategies perform, we indirectly compare them using random tuning as a baseline. In particular, we execute each tuning algorithm and count the number of iterations n until it stops and returns its best value A. Then, we randomly sample same number of values n from the search space and determine the best value B. The relative difference between A and B provides a first insight how well the tested algorithm optimizes. To exclude bias, we run each experiment 500 times, which leads to stable convergence results within 95% confidence intervals.

Table 1 presents the average relative improvement of each tuning algorithm's result in comparison to random tuning, on the same number of respective evaluations as described above.

The table surprisingly reveals that Simplex and Polytope have worse tuning results than Random. That is, Simplex would provide on average a program run-time that is 3.5% worse and Polytope a run-time that is 11.4% worse than Random, i.e., if the same number of random configurations were chosen in each experiment in the same context. Basic Evolution as well as Balanced and Unbalanced Evolution find better-performing configurations than Random. Balanced Evolution ranks best. These aggregated averages provide a high-level overview, but miss details when it comes to understanding the tradeoff between number of evaluations and optimization results. To gain this deeper insight, we conduct additional analyses in (c) and (d) to paint a more precise picture.

Tuning Strategy	Avg. Improvement over Random
Balanced Evolution	18.5%
Unbalanced Evolution	18.1%
Basic Evolution	12.3%
Particle Swarm	8.5%
Differential Evolution	1.8%
Simplex	-3.5%
Polytope	-11.4%

Table 1: Comparison of tuning strategies with random tuning.

7.1.3 Trade-off Analysis

Figure 4 presents another perspective. It shows for each tuning strategy the trade-off between the number of evaluated configurations (y-axis) vs. the tuning error relative to the best algorithm. We compute the error as the average difference between the best returned value by an algorithm and the global optimum (which is known because the functions are known). The relative tuning error positions each algorithm in comparison to the best algorithm, i.e., the one that got closest to the global optimum. Vertical bars illustrate the standard deviation of evaluated configurations. The width of horizontal bars shows the 95% confidence interval for the mean error. Algorithms in the lower half of the graph need fewer program evaluations, whereas algorithms in the left half have better optimization results.

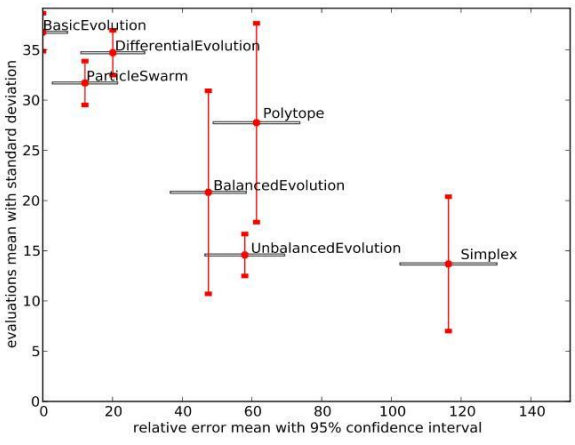


Figure 4: Trade-off comparison: Number of evaluations vs. optimization error.

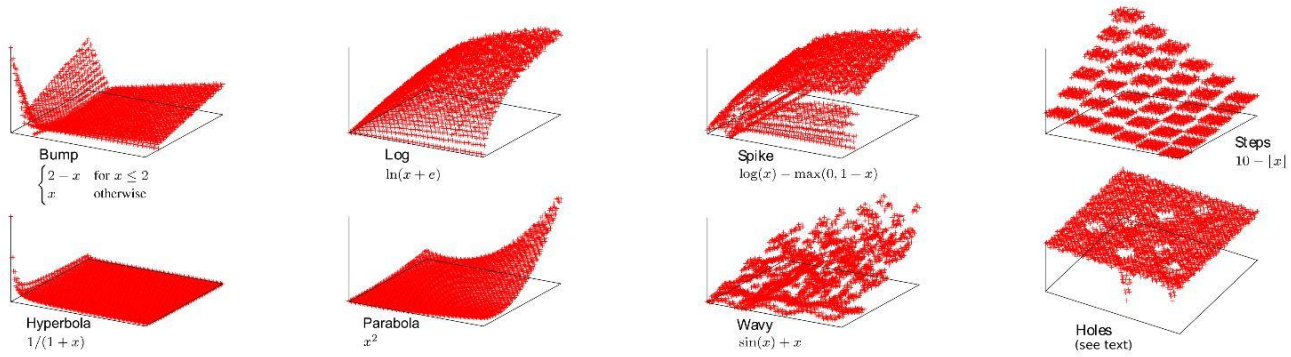


Figure 3: Excerpt of search space shape models that are visualized in 3 dimensions.

Evolutionary algorithms have significantly lower errors than Simplex. Basic Evolution optimizes best and Simplex worst. One could hypothesize that the bad result for Simplex is due to fewer iterations, however, Polytope shows that additional iterations do not reduce errors significantly. Among the evolutionary approaches, Unbalanced Evolution requires the fewest iterations and still beats Polytope and Simplex in finding configurations closer to the optimum. Particle Swarm optimizes second-best, but as shown later, it does not work well on real programs, but evolutionary algorithms do.

7.1.4 Impact of Search Space Shape on Tuning

Figure 5 answers the question what impact the different search space characteristics have on each tuning strategy. The Figure plots the average tuning error (computed from 500 trials averaging the absolute differences between the best value found and the actual optimum) for each model, which should ideally be zero.

The bars show that all algorithms find values close to the optimum on the Hyperbolic function, where large regions of the search space have values close to the global optimum. All algorithms also work well on the Parabolic function. Simplex-based tuning does not work well on the functions Wavy, Steps and Spike. This observation suggests that it will not work well on programs with noisy or erratic performance behavior. However, the Polytope extension is capable of compensating some of Simplex' weaknesses. All algorithms fail on the Holes function (bars are cut-off in the graph), which is a tough case, however, Basic Evolution leads the field there as well, whereas Polytope and Simplex are last (2.26x worse than Basic Evolution). Performance tuning will likely be inefficient with any algorithm if program performance can only be characterized by the Holes function.

7.2 Tuning Analysis of Real Programs

7.2.1 Benchmark

We evaluate all tuning algorithms on parallel programs from the widely used PARSEC [2] benchmark suite, which is composed of thirteen multithreaded shared-memory programs aimed at representing a broad spectrum of workloads on today's multicore systems. PARSEC includes programs such as video encoding, image processing, ray tracing, clustering, data mining, simulations, content search, compression, and others. Our experiments are carried out on the following multicore platforms:

- *4-core machine.* Intel Core2 Quad CPU with 4 cores, 2.4 GHz, 3GB RAM, Ubuntu Linux 10.04 with kernel 2.6.32.

- *8-core machine.* Intel 8-core machine with 2x Quadcore Xeon E5320 processor, 1.86 GHz, 8 GB RAM, Ubuntu Linux 10.4 with kernel 2.6.32.

We conduct a quantitative tuning effectiveness study for our combination of multicore programs and systems. We exhaustively sample the entire search space and determine the global optimum run-time for each configuration, so we can compute the error of our algorithms compared to the true optimum of each application. Then, each tuning algorithm walks through the same search space based on one machine and input data set, so there is a fair comparison regarding the number of required evaluations and tuning error. Each algorithm runs to completion 500 times for each PARSEC program, which provides acceptable results within 95% confidence intervals.

7.2.2 Inputs

There are two input data sets: "medium" size (PARSEC "simlarge") and "large" size (PARSEC "native"). The "medium" set leads to execution times of about 12–20 *seconds* for one run for one program; for the "large" set it is approximately 10–30 *minutes*. We exhaustively execute all program configurations for each thread number (1..60), machine (1,2), and program (13); it takes over a month alone to compute the data for all $60 \times 2 \times 13 = 1560$ configurations, which is why this experimental evaluation is limited to the thread parameter (but complemented with more parameters in the model-based evaluation of the previous Section). We conduct experiments to find the number of threads for which each program has the lowest run-time on each platform. Tuning the thread parameter manually would already have non-intuitive outcomes; for example, the *vips* workload has its optimum of 56 seconds at 22 threads on our 8-core machine, whereas 8 threads have a runtime of 67 seconds.

7.2.3 Results

Figure 6 compares the tuning results of each algorithm for each platform and data set. Here, we focus on this tradeoff analysis that shows the key results. In each graph, the left-most algorithm has the lowest tuning error, and the right-most the highest. Algorithms closer to the bottom of each graph require fewer configuration evaluations, i.e., they will

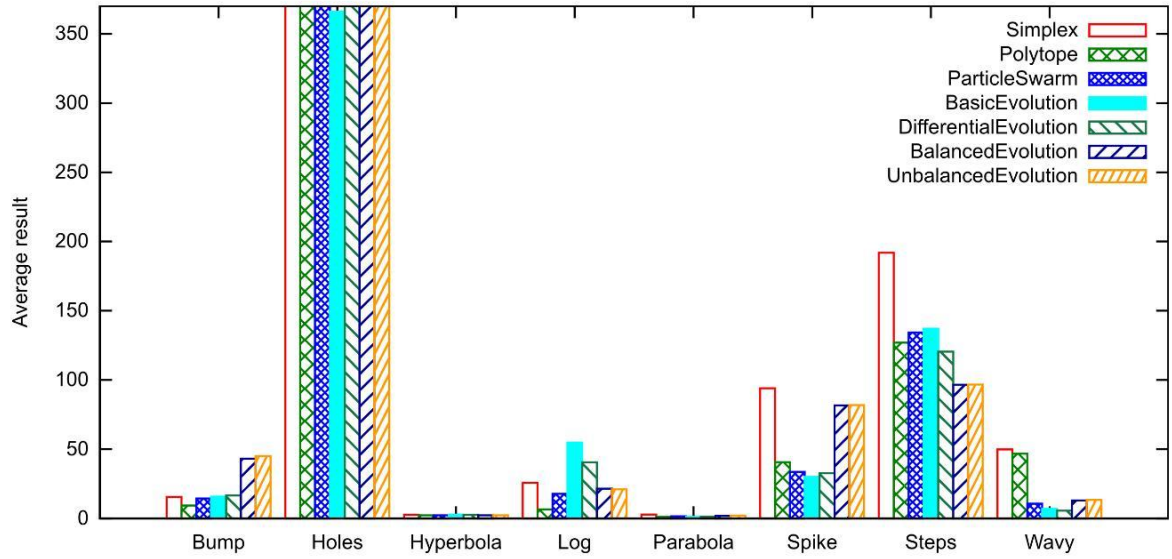


Figure 5: Impact of search space shape on average tuning error (0 is best result).

execute a tunable program less often. For each tuning strategy, a circle in the middle of a vertical bar indicates the average number of iterations. The width of horizontal bars shows the 95% bootstrap confidence interval for the mean error. A vertical line indicates the standard deviation in the number of total evaluations (using the coefficient of variation in Figures 4 and 6 leads to smaller bars, but the same conclusions).

4-core platform. On the medium data set, Figure 6(a) shows that the evolutionary algorithms have lower errors (i.e., they find better performance configurations) than all other algorithms. Balanced Evolution is the best, followed by Basic Evolution and Differential Evolution. Particle Swarm ranks last, being almost 80% worse than Balanced Evolution. It is worth noting that Simplex beats Particle Swarm with a lower error using almost the same average number of evaluated configurations, but the optimization error is higher than that of evolutionary algorithms. On the large data set, Figure 6(b) shows that Basic Evolution has the lowest error, followed by Differential Evolution. Basic Evolution and Balanced Evolution have a similar average evaluation counts, however, Basic Evolution still has a lower error. Last ranked is Particle Swarm, which is almost 60% worse than Basic Evolution. Simplex is second-last, being more than 50% worse than Basic Evolution.

8-core platform. On the medium data set, Figure 6(c) shows that Basic Evolution ranks first (with the lowest error) followed by Differential Evolution. Particle Swarm ranks last, being over 400% worse than Basic Evolution, and Simplex is second-last being 320% worse. Polytope ranks third; its strategy can obviously compensate in this context the shortcomings of the Simplex approach, so visiting simultaneously more points in the search space pays off. On the large data set, Figure 6(d) shows that Basic Evolution ranks first, again followed by Differential Evolution. Simplex, Polytope, and Particle Swarm are on the last ranks.

7.3 Discussion and Insights

Evolutionary approaches such as Basic Evolution have consistently lower tuning errors than the other approaches. Evidence shows that application tuning is influenced by the input data size and the characteristics of each platform, but that the evolutionary strategies adapt well.

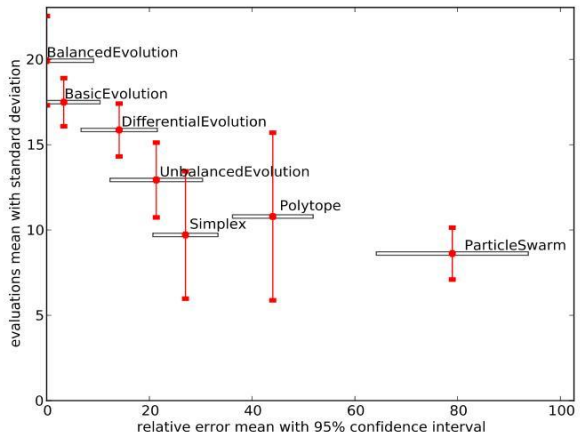
In three out of four times, Basic Evolution ranks first and Differential Evolution ranks second. Balanced evolution ranks first once. Unbalanced Evolution often gets stuck in local minima, due to its design, but is still better than others. At the other end, Particle Swarm ranks last in three out of four times, and the other algorithms often produce better results for a similar number of program evaluations. Evolutionary approaches have lower errors than Simplex which ranks third-last two times and second-last two times. Evolutionary algorithms typically need slightly more evaluations, but lead to better results. Polytope shows that extending Simplex to visit more points does not help much, as its search rules do not match well to typical multicore workloads.

The advantage of evolutionary tuning strategies is that they have an inherent, continuously executed randomization that complements their systematic search. This randomization allows them to better cope with noise and rocky shapes of search spaces that trap the other algorithms into local minima. This effect has been confirmed by our observations in the model-based evaluations as well as for real multicore applications.

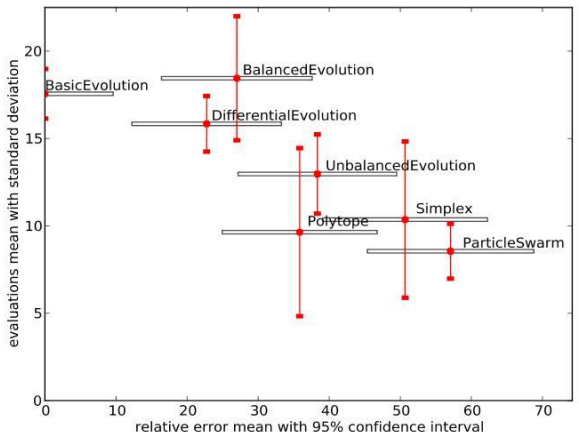
Our results suggest that general-purpose auto-tuners that aim to optimize a variety of application parameters of different types of applications do well with evolutionary tuning strategies.

8. RELATED WORK

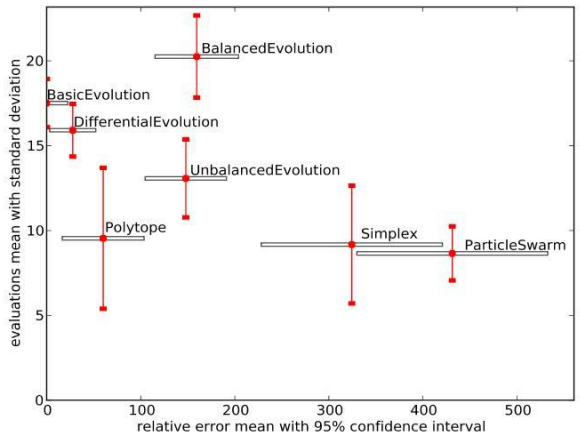
Feedback-directed performance optimization has been investigated in various contexts. Auto-tuning approaches are predominantly applied in numerics and typically generate the platform-specific code of an entire application (e.g., ATLAS [24] and OSKI [22] for matrix computations, FFTW [8] for FFT, FIBER [9] for eigensolvers, and SPIRAL [14] for DSP). In [3], ORIO is used as a code annotation and transformation tool to generate different versions of numerical kernel codes, based on various tuning parameters that are tuned using genetic, simplex, and random algorithms in Matlab. By contrast, we do not require a particular type of application. We can tune various kinds of applications, including non-numerical ones, which other auto-tuners do not apply to at all. We focus on application-level parameters and thus



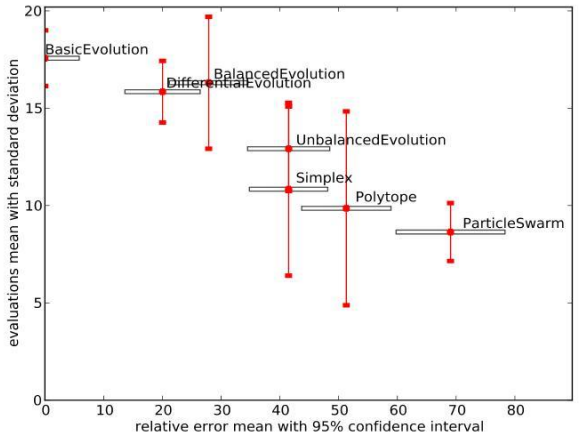
(a) Medium data set, 4-core platform



(b) Large data set, 4-core platform



(c) Medium data set, 8-core platform



(d) Large data set, 8-core platform

Figure 6: Tuning comparison on the PARSEC benchmark on 4-core and 8-core platforms.

do not generate the code of an entire application. We do not exclude, however, that programmers can integrate components of numerical kernels that are pre-tuned with other tuners. Our goal is fundamentally different from numerical application tuning: We aim to support software engineers to optimize all sorts of parameters and automate the search process. This direction contrasts other proposals targeting lower-level compiler optimizations [1, 5, 18, 21, 23, 25], but which can be also used in combination with our approach. [17] also compares search algorithms, however, just on single-threaded performance and merely on four search spaces that stem from two dense linear algebra routines. The results show that particle swarm optimization is good for tuning loop unrolling and blocking in that context; by contrast, our work shows that particle swarm optimization does not work well for tuning multicore application parameters in a broader spectrum of applications. [16] uses fuzzy rules for reactions to resource changes in grids. [20] requires that programmers describe tuning options in a proprietary resource specification language, and it employs a simplex-based algorithm for tuning. Our report, however, shows that simplex-based algorithms might not be efficient and that evolutionary algorithms lead to tuning results with lower errors.

9. CONCLUSION

Multicore application performance tuning is difficult. Due to the variety of available hardware, custom-tuned parallel programs might perform well on one platform, but poorly on others. To remain portable, multicore applications have to be adaptive. This report presents a domain-independent approach to configure application-level performance parameters efficiently and facilitate the achievement of good performance on different multicore platforms. Our feedback-directed optimization relieves software engineers from the tedious and inefficient trial-and-error search that is often pursued due to lack of smarter automation. As demonstrated, our technique works well for a variety of domains and programs including video encoding, image processing, ray tracing, clustering, data mining, simulations, content search, compression, and others. We overcome a major constraint of previous approaches, which typically focused auto-tuning on just one type of program. Identifying working search strategies that apply to breadth rather than depth is not trivial. We investigate several search strategies and provide a series of quantitative results based on stress-test models and real programs. The results clearly show that evolutionary strategies find the best performance configurations and beat other tuning strate-

gies that are commonly used in the literature. The specific combination of systematic and randomized search is a key factor why evolutionary approaches are superior in our context. This key finding provides a fertile ground for future auto-tuning research to pursue general-purpose performance tuning based on evolutionary strategies.

10. REFERENCES

[1] Ctuning project. http://ctuning.org, 2011.

[2] The PARSEC benchmark suite. http://parsec.cs.princeton.edu, 2011.

[3] P. Balaprakash et al. Can search algorithms save large-scale automatic performance tuning? Technical report ANL/MCS-P1823-0111, Argonne National Laboratory, January 2011.

[4] R. R. Barton and J. S. Ivey, Jr. Modifications of the nelder-mead simplex method for stochastic simulation response optimization. In *Proc. WSC '91*, 1991.

[5] C. Chen et al. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *Proc. CGO '05*, 2005.

[6] K. A. De Jong. *An analysis of the behavior of a class of genetic adaptive systems.* PhD thesis, Ann Arbor, MI, USA, 1975.

[7] B. Efron. R.J. Tibshirani. *An introduction to the bootstrap.* New York: Chapman & Hall, 1993.

[8] M. Frigo and S. Johnson. Fftw: an adaptive software architecture for the FFT. In *Proc. IEEE ICASSP'98*, volume 3, pages 1381–1384, 1998.

[9] T. Katagiri et al. Fiber: A generalized framework for auto-tuning software. In *Proc. ISHPC*, 2003.

[10] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proc. IEEE Int. Conf. on Neural Networks*, Piscataway, NJ, 1995.

[11] Z. Michalewicz and D.B. Fogel. How to Solve It: Modern Heuristics. Springer Verlag, 2004.

[12] V. Pankratius. Search Algorithms for Automatic Performance Tuning of Parallel Applications on Multicore Platforms. *Technical Report 2010–8*, Karlsruhe Institute of Technology, Institute for Program Structures and Data Organization, July 19, 2010.

[13] V. Pankratius and W.F. Tichy. Truck Scheduling on Multicore. *it-Information Technology Journal* 53(2), pp. 60-65, Oldenbourg, 2011.

[14] M. Puschel et al. Spiral: code generation for dsp transforms. *Proc. of the IEEE*, 93(2), 2005.

[15] I. Rechenberg. *Evolutionsstrategie: Optimierung Technischer Systeme nach Prinzipien der biologischen Evolution.* Frommann-Holzboog, 1973.

[16] R. Ribler et al. Autopilot: Adaptive control of distributed applications. In *Proc. IEEE HPDC*, 1998.

[17] K. Seymour et al. A Comparison of search heuristics for empirical code optimization. In *Proc. CGO*, 2008.

[18] M. Stephenson et al. Meta optimization: improving compiler heuristics with machine learning. In *Proc. PLDI'03*, 2003.

[19] R. Storn and K. Price. Differential evolution- a simple and efficient adaptive scheme for global optimization over continuous spaces. Technical report, 1995.

[20] C. Tapus et al. Active harmony: Towards automated performance tuning. In *Proc. HPNC*, 2003.

[21] A. Tiwari et al. A scalable auto-tuning framework for compiler optimization. In *Proc. IPDPS'09*, pages 1–12, 2009.

[22] R. Vuduc et al. Oski: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16(1):521+, 2005.

[23] Z. Wang and M. F. O'Boyle. Mapping parallelism to multi-cores: a machine learning based approach. In *Proc. PPoPP'09*, 2009.

[24] C. R. Whaley et al. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1-2):3–35, January 2001.

[25] K. Yotov et al. Is search really necessary to generate high-performance blas? *Proc. of the IEEE*, 93(2):358–386, February 2005.