AutoTunium: An Evolutionary Tuner for General-Purpose Multicore Applications

Andreas Zwinkau
zwinkau@kit.edu
Karlsruhe Institute of Technology (KIT), Germany

Victor Pankratius
pankratius@csail.mit.edu
Massachusetts Institute of Technology (MIT), USA

***Abstract*—Today's increasing diversity in multicore hardware challenges programmers when it comes to software performance optimization and portability. As multicore processors are in almost every PC and server, programmers now have to parallelize a larger spectrum of applications, many of which are non-numerical. To obtain good performance, programmers typically try out different software tuning parameter configurations on each platform. However, this manual approach to finding good configurations in the search space is impractical due to combinatorial explosion, but yet it is common practice due to lack of alternatives for general programs. This paper presents a smarter way to tackle this problem algorithmically for a variety of multicore applications, including non-numerical ones. Our work introduces AutoTunium, a novel feedback-directed optimizer that automates the application tuning process with evolutionary search strategies. The software infrastructure is easy to use and integrated in the popular Eclipse environment. It collects run-time information to predict parameter configurations that are likely to lead to good performance in future runs, and configures programs for production runs in the best possible way. We quantify the effectiveness of various tuning strategies on a diverse set of real applications and multicore platforms. The evaluation shows that AutoTunium's evolutionary strategies work well despite the broad scope of applications and perform better in this context than other simplex-based search algorithms. Our insights are derived from model-based analyses as well as from performance analyses with real programs in the PARSEC benchmark suite.**

Keywords: Multicore; performance tuning; portability

I. Introduction

Multicore processors with several cores on a chip are standard, and programmers are now challenged to parallelize all kinds of performance-critical applications. A problem that makes multicore programming hard is that multicore platforms are different, e.g., with respect to 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.

Automatic performance tuning is promising in this context, but existing techniques focus on particular domains in numerics, such as FFT, signal processing, and matrix multiply [9], [10], [13], [23]. Moreover, low-level compiler optimizations often miss important leverage for performance that could have been additionally achieved by tuning in higher abstraction layers [1], [17], [22]. Because of this focus, numeric kernel tuners are not designed to work for a wider spectrum of programs, especially for ones that do not have any numerical kernels. Unfortunately, many of today's multicore applications on desktops and servers fall into this category. Due to lack of alternatives, software engineers tackle this problem with a largely manual approach. First, they introduce changeable tuning parameters (i.e., "tuning knobs") in their software, such as number of threads in application thread pools, buffer sizes, maximum number of workers, size of data partitions, choices for algorithms, etc. Then, they try out different parameter values to find the ones that yield the best performance on each platform. It is obvious that for k parameters the entire search space is cross product of all domains, $dom(p_1) \times dom(p_2) \times \ldots \times dom(p_k)$. Exhaustive search is unrealistic because every tuple evaluation requires at least one program run, which could last hours. Also, intuition can be a false friend; for example, applications could miss speedup opportunities where more threads hide latency, whereas applications that increase synchronization overhead with more threads would slow down.

This paper shows a smarter and more efficient way to tackle this problem. It makes the following novel contributions. It introduces AutoTunium, a new extensible tuning infrastructure that works in the Eclipse development environment. AutoTunium is designed to work with different sorts of multicore applications and is demonstrated, among others, on video encoding, image processing, ray tracing, clustering, data mining, simulations, content search, and compression. AutoTunium uses a tuning technique that works well in a breadth of domains, rather than in one single domain as previous work in [9], [10], [13], [23]. AutoTunium combines systematic search and randomized search to escape local minima and provides the means for optimizations beyond fine-granular instruction optimizations [1], [17], [22]. We provide a thorough evaluation comparing variants of evolutionary tuning, simplex-based tuning, swarm tuning, and random tuning, using model-based analyses as well as benchmarks with real programs from the PARSEC [2] benchmark suite. We show that AutoTunium's evolutionary search outperforms simplex approaches that are most commonly used on other platforms [19].

The paper is organized as follows. Section II details the optimization problem. Section III sketches the AutoTunium performance tuning framework. Section IV presents the pluggable evolutionary tuning strategies. Sections V and VI describe particle swarm tuning, simplex- and polytope-based methods as alternative and commonly used tuning. Section

VII evaluates and compares all tuning techniques. Section VIII contrasts related work. Section IX provides our conclusion.

II. The Problem

Our particular optimization problem is related to offline tuning and can be formulated as follows. Given a multicore program P with k performance-relevant parameters (assumed to be accessible via command line), the goal is to minimize P's execution time iteratively. We start with an initial parameter configuration, then execute P, measure run-time, and calculate 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 runs of the program. After the tuning process ends, the program uses the best configuration found so far.

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 our approach is based on empirical search methods [12] that don't require derivatives and use just function evaluations.

As evolutionary search methods use randomization, result comparison is tricky. We evaluate tuning strategies with two common empirical metrics. The 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), motivated by the fact that run-times can vary for different programs in various domains, from seconds to days. If the number of iterations was fixed, programs with short run-times might 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. The number of evaluations is therefore more useful for a cross-comparison of tuning strategies rather than just looking at the sum of program execution times plus some overhead between runs. In our scenarios, parameter reconfiguration occurs between program runs, so tuning time and tuning overhead does not affect program run-time, and tuning overhead is typically dwarfed by P's execution time.

III. AutoTunium: A Software Infrastructure For Multicore Application Performance Tuning

AutoTunium is a multicore performance tuning infrastructure that can be connected to any type of executable program; it also does not require programmers to write their applications in a particular language. These features greatly relax many constraints imposed by previous work (see Section VIII). AutoTunium is aimed at average developers who need to tune multithreaded applications with many performance-relevant parameters (typically accessible on the command line). Figure 1 shows parts of the user interface, which is integrated into the Eclipse development environment.

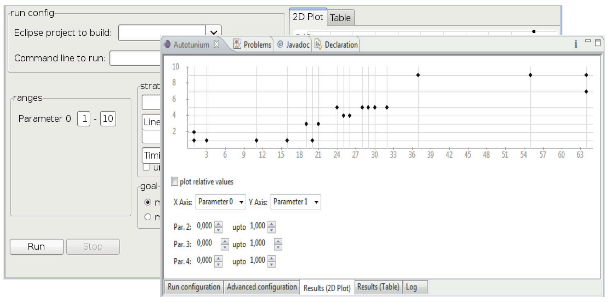


Figure 1. Screenshot of AutoTunium in Eclipse.

AutoTunium is written in Java and is extensible with tuning plugins that are implemented in Java. Plug-ins do not require recompilation and can be treated as scripts. AutoTunium comes along with several tuning plugins, but new plugins could be downloaded for example from Web repositories when programmers want to update or customize search strategies for different sets of programs. All algorithms in this paper are implemented as plugins, so they all use the same common tuning infrastructure. Also, all techniques are carefully ensured to work on a discrete space.

AutoTunium optimizes as described in the previous Section. Program run-time is optimized using multi-dimensional search algorithms and user-defined constraints for parameter values. The objective function used here is minimizing program run-time, though it is possible to specify other minimization and maximization problems.

IV. AutoTunium's Evolutionary Tuning

AutoTunium's evolutionary tuning strategies are inspired by [14], [18]. In the context of general-purpose multicore application tuning these techniques have not been explored thoroughly so far; this paper thus conducts a detailed analysis and compares various adaptations in the context of multicore performance tuning.

A. 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 [14]. 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.

In each generation, AutoTunium creates 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.

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 $\alpha b + \beta s + \gamma r$
```

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.

B. 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 $i \in \{1, \dots, |x|\}$ do
        $z_i = \begin{cases} x_i & \text{with probability } \alpha \\ y_i & \text{else} \end{cases}$
    return $z$
```

The mutation operator [18] picks four random vectors $p_1, \dots, p_4$. It scales the difference $p_2 - p_3$ by a differential weight factor $F \in]0, 2]$ and updates $v = p_1 + F \cdot (p_2 - p_3)$. 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 is added to the population p, and the individual with the worst fitness value is removed.

C. Balanced Evolution

AutoTunium's balanced evolution is a new technique that initializes starting configurations with 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 $\{(1, 1), (n, 1), (1, m), (n, m)\}$. Other random configurations are added until the initial population has the same size as in the other presented algorithms, to keep results comparable.

Algorithm 3 Balanced Evolution Tuning

```
$p \leftarrow$ starting population
for $g$ generations do
    $p \leftarrow p \cup \{\text{MUTATION}(p)\}$
return $\text{selection}_1(p)$

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

The point to evaluate next in the search space is selected based on the following rationale. On the one hand, uncertainty should be reduced by avoiding that parts of the search space are not covered at all. On the other hand, we need focus around configurations that are promising to be a global optimum. We assume that points with good performance are in the neighborhood of other points with good performance so far.

We associate $i_p = \textit{uncertainty} - \textit{potential}$ to a point $\vec{p}$ in the search space. In particular, $\textit{uncertainty} = |\vec{n}_p - \vec{p}|$ and $\textit{potential} = (t(\vec{n}_p) + t(\vec{n}'_p))/2$, where t is run-time, $\vec{n}_p$ is the nearest evaluated point to $\vec{p}$, and $\vec{n}'_p$ the second nearest. $\textit{uncertainty}$ steers the coverage of the search space, whereas $\textit{potential}$ steers search convergence. When generating a new individual, AutoTunium 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 discard data learned so far. In extremely rare cases where $\vec{n}_p$ and $\vec{n}'_p$ are next to each other and the population does not change, AutoTunium starts over.

D. Unbalanced Evolution

In a variant of balanced evolution AutoTunium 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.

V. Particle Swarm Tuning

Particle swarm tuning [11] works in a similar way as evolutionary tuning. In principle, AutoTunium uses particles floating through the search space with a certain inertia and which are expected to flock eventually around the global minimum.

Algorithm 4 Particle Swarm Tuning

```
initialize particle swarm $S$
$b \in S$ with $t(b)$ minimal
for $k$ steps do
    for $p \in S$ do
        //$p^*$ is the best in $p$'s history
        //$d_p$ is $p$'s current movement
        //vector
        $d_p \leftarrow d_p + \alpha(b - p) + \beta(p^* - p)$
        $p \leftarrow p + d_p$
        if $t(p) > t(b)$ then
            $b \leftarrow p$
return $b$
```

Autotunium's 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.

The particle count is logarithmic in relation to the search space size, and step count is linear in dimension size. AutoTunium terminates after a predefined step count. This design makes results comparable to the ones discussed in the other sections.

VI. Simplex- and Polytope-Based Tuning

For comparison purposes, AutoTunium also has a plugin for the well-known Nelder-Mead technique [5] that works with 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). 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.

The 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. Autotunium extends the simplex technique to work for polytopes, as a simplex is a special case of a polytope that has $s \in (\mathbb{R}^d)^x$ and $x > d + 1$ compared to a simplex. Here, AutoTunium assumes that more points will improve tuning quality.

Two factors have to be balanced; on the one hand, x should be large, which implies that there is more information when making a decision. On the other hand, x should not be too large

Algorithm 5 Simplex-based Tuning

```
$s \in (\mathbb{N}^n)^{n+1}$
$n \leftarrow \text{reflexion}_\alpha(s)$
while $u > |s| \cdot d$ do
    if $t(n) > t(\text{worst}(s))$ then
        $n \leftarrow n + \beta \cdot (\text{best}(s) - n)$
        if $t(n) > t(\text{worst}(s))$ then
            $\text{compress}_\beta(s)$
            $n \leftarrow \text{reflexion}_\alpha(s)$
    else
        if $t(n) < t(\text{best}(s))$ then
            $n \leftarrow n + \gamma \cdot (n - \text{worst}(s))$
        else
            $s \leftarrow (s \cup \{n\}) \setminus \{\text{worst}(s)\}$
            $n \leftarrow \text{reflexion}_\alpha(s)$
return $m$
```

and cause a large number of initial evaluations, which would render the approach too expensive due to repeated program executions. Our evaluations revealed that there is no significant difference for $x \in \{2d, 4d, 8d\}$, so AutoTunium initializes polytopes with $x = 4d$, while applying the same optimization rules as in simplex-based optimization.

VII. How Does AutoTunium Perform?

This Section evaluates AutoTunium's tuning strategies from several perspectives: (1) in a model-based approach with known and complex multidimensional search spaces, and (2) on a suite of real parallel programs that are tuned on 4-core and 8-core platforms.

Starting with search space models has the advantage to 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 practical problem is that the number of iterations (i.e., how often they execute a program that is tuned) cannot always be controlled for every tuning strategy. 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 developer whether he or she favors fewer iterations or lower run-times. This is why we conduct several analyses from different perspectives to quantify these tradeoffs. We employ percentile boostrapping [8] to estimate confidence intervals and make sure that our results are within acceptable ranges.

A. Model-Based Analyses

*1) **Program Performance Models**:* We start with models of multicore program performance to analyze the tuning strategies in a controlled environment. De Jong [7] 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 2; 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 [7]. All functions are discretized and scaled to $[1, 1000]$. Up to 10% of noise is added to simulate fluctuations of real measurements.

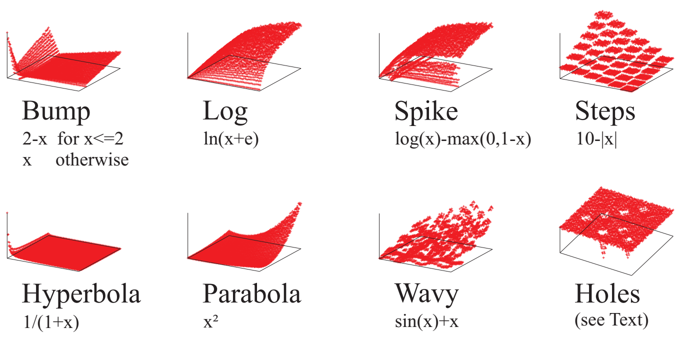


Figure 2. A 3-D Example of Multicore Performance Search Spaces.

*2) **Overall Comparison**:* This initial comparison uses 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.

The following table presents the average relative improvement of each tuning algorithm's result in comparison to random tuning, on the same number of respective evaluations.

Tuning Strategy	Avg. Improvement over Random Tuning
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%

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. Even though these averages provide a high-level overview, they miss details when it comes to understanding the tradeoff between number of evaluations and optimization results. We therefore conduct additional analyses, as shown next.

*3) **Trade-off Analysis**:* Figure 3 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.

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, where evolutionary algorithms do.

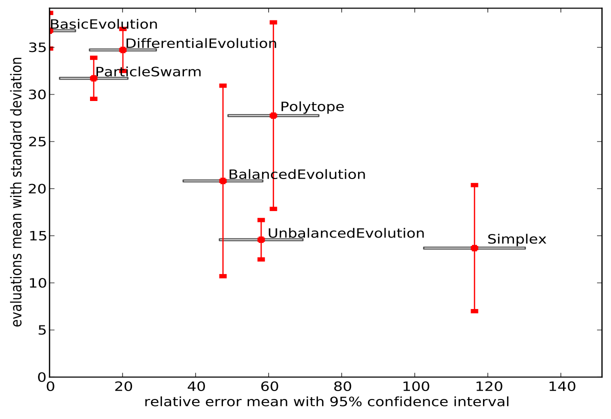


Figure 3. Trade-off comparison: Number of evaluations vs. optimization error.

*4) **Model Impact on Tuning**:* Figure 4 shows what impact the different function shapes 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.

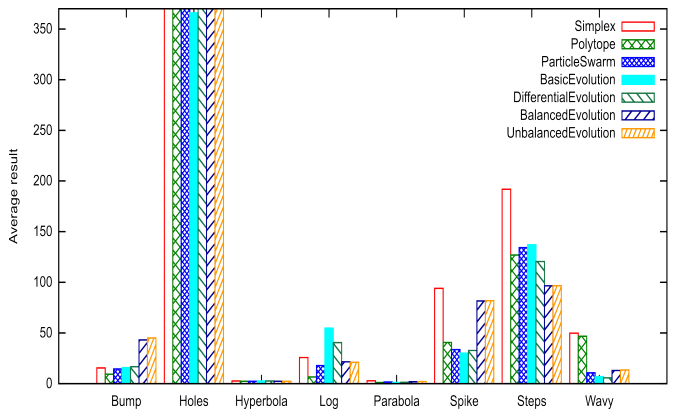


Figure 4. Impact of model on tuning effectiveness.

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, while Polytope and Simplex are last (2.26x worse than Basic Evolution). Thus performance tuning will likely be inefficient with any algorithm if program performance can only be characterized by the Holes function.

B. Tuning Analyses with Real Programs

The practical experiments complement the model-based evaluation presented in the previous Section in real-world scenarios.

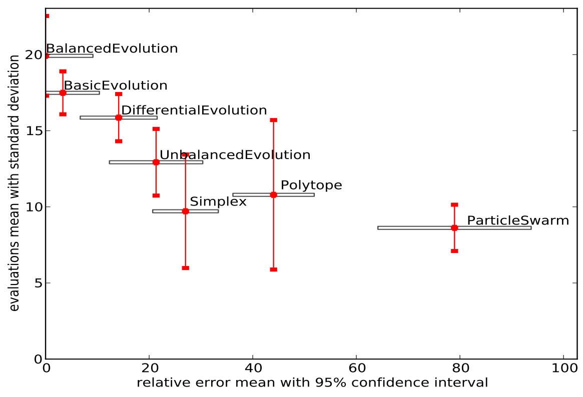
*1) **Benchmarks**:* We evaluate all tuning algorithms on parallel programs from the widely used PARSEC [2] benchmark suite, which is composed of thirteen multithreaded shared-memory programs aimed at representing a broad spectrum of workloads on today's multicore systems. PARSEC includes programs such 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: (1) Intel Core2 Quad Q6600 CPU with 4 cores, 2.4 GHz, 3GB RAM, Ubuntu Linux 10.04 with kernel 2.6.32. (2) 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. Our setup allowed us to gather over a longer period of time the entire search space and determine the global optimum run-time for each application configuration. This way we can compute the error of our algorithms compared to the true optimum for each application on each platform. Then, each tuning algorithm walks through the same search space on of the respective machine and input data set, to enable a fair comparison on 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.

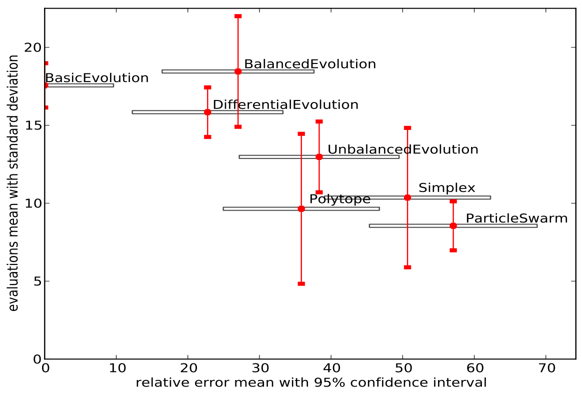
*2) **Inputs**:* The evaluation uses two input data sets: "medium" size (PARSEC "simlarge") and "large" size (PARSEC "native"). The "medium" set leads to execution times of about 12–20 *seconds* for one run for one program; for the "large" set it is approximately 10–30 *minutes*. We exhaustively execute 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 one performance parameter (and complemented with more parameters in the model-based analysis in the previous Section). Here, the experiments determine the optimum number of threads to use on each platform; even though this might appear easy, this parameter already has non-intuitive outcomes. For example, the *vips* workload has its optimum at 22 threads on our 8-core machine, and not at 8 threads as one might intuitively expect. Actually, the runtime at 8 threads is 20% worse than the best achievable runtime at 22 threads.

*3) **Results**:* Figure VII-B compares the tuning results of each tuning strategy for each platform and data set. Due to space limitations, we just graph 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 execute a tunable program less often. For each tuning strategy, a filled circle indicates the average number of iterations and the resulting error. The width of horizontal bars shows the 95% confidence interval for the mean error. A vertical line indicates the std. deviation in the number of total evaluations (remark: using the coefficient of variation in Figures 3 and VII-B yields smaller bars, but leads to the same conclusions).

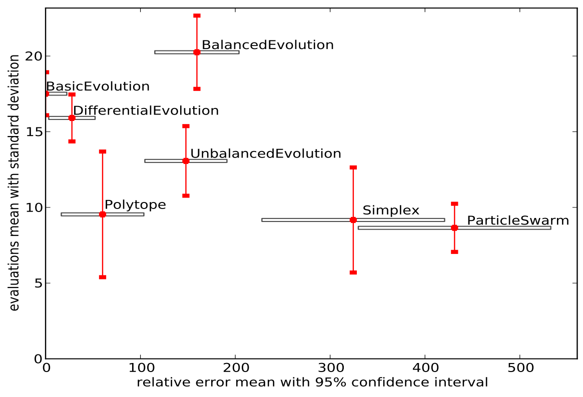
4-core platform. On the medium data set, Figure 5(a) shows that the evolutionary algorithms have lower errors (i.e., find better performance configurations) than all other algorithms. Balanced Evolution is the best, followed by Basic Evolution and Differential Evolution. Particle Swarm ranks last, being 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 5(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



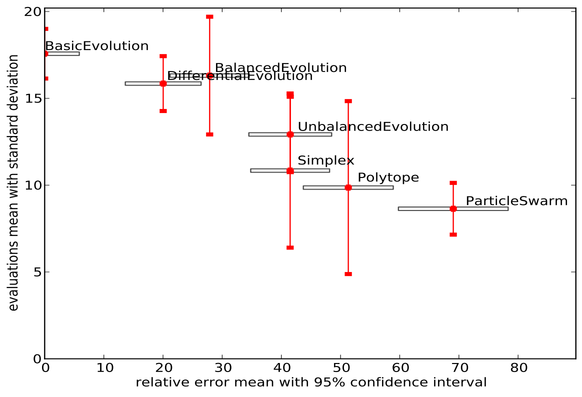
(a) Medium data set, 4-core platform



(b) Large data set, 4-core platform



(c) Medium data set, 8-core platform



(d) Large data set, 8-core platform

Figure 5. Tuning comparison on the PARSEC benchmark on 4-core and 8-core platforms.

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 5(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 5(d) shows that Basic Evolution ranks first, again followed by Differential Evolution. Simplex, Polytope, and Particle Swarm are on the last ranks.

C. *Discussion and Insights*

AutoTunium's evolutionary strategies work well for general-purpose parameter tuning in our diverse set multicore applications. Evolutionary approaches such as Basic Evolution have consistently lower tuning errors than the other approaches. The data 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.

VIII. Related Work

Approaches applying auto-tuning are predominantly used in numerics and typically generate the platform-specific code of an entire application (e.g., ATLAS [23] and OSKI [21] for matrix computations, FFTW [9] for FFT, FIBER [10] for eigensolvers, and SPIRAL [13] for DSP). In [4], ORIO is used as a code annotation and transformation tool to generate different versions of numerical kernel codes. To take advantage of all tuning features, the aforementioned techniques and others such as [3], [19] usually require developers to program the whole application in proprietary language or use some proprietary resource specification language.

AutoTunium's approach differs in several important ways. It does not require using a particular programming language, but merely needs an exposition of application tuning parameters to the tuner. AutoTunium also does not concentrate on solving one particular problem, such as matrix multiply, but extends to evaluating the performance configuration search space of a variety of general-purpose multicore applications. Another difference is that AutoTunium does not generate the entire code of the tunable application in each iteration, but instead reconfigures an existing application. It is not excluded, however, that AutoTunium can be used in conjunction with other optimizers or compiler-level tuners such as [1], [6], [17], [20], [22], [24].

With respect to tuning, other works such as [19] require that programmers describe tuning options in a proprietary resource specification language, which is not necessary in AutoTunium. The system in [19] also employs a simplex-based algorithm for tuning, however, our results show that such techniques should not be the first choice when configuring a set multicore programs from a variety of fields. The work of [15] uses fuzzy rules for an adaptive control approach in non-multicore systems. Other comparisons in [16] are limited for single-threaded performance on four search spaces that stem from two dense linear algebra routines; in that context particle swarm optimization was good for tuning loop unrolling and blocking. By contrast, particle swarm optimization does not work well in our context for tuning multicore application parameters.

Overall, AutoTunium has a broader scope to support software engineers in general-purpose multicore application development and tuning parameter configuration.

IX. Conclusion

Performance tuning of multicore applications has become difficult due to the hardware variety. Non-adaptive multicore software might thus perform well on one platform, but poorly on others. The lack of practical solutions for the tuning of general-purpose multicore applications traps many software engineers into tedious trial-and-error processes with large search spaces. This paper presents a smarter way of configuring a multicore application's tuning knobs with an algorithmic approach. The AutoTunium system demonstrates the applicability of automatic tuning on a wide set of different programs including video encoding, image processing, ray tracing, clustering, data mining, simulations, content search, and compression. AutoTunium's evolutionary tuning strategies find the best performance configurations and outperform other commonly used strategies in literature. The specific combination of systematic and randomized search is a key factor why evolutionary strategies are superior in our context. Our system also overcomes a major constraint of previous solutions that apply only to specific numerical kernels. Overall, AutoTunium makes an important leap not only towards better multicore performance, but also towards better portability and increased programmer productivity.

Acknowledgements

We thank the German Excellence Initiative and KIT for their support while the second author was at KIT in Karlsruhe.

References

[1] Ctuning project. http://ctuning.org, 2011.
[2] The PARSEC benchmark suite. http://parsec.cs.princeton.edu, 2011.
[3] J. Ansel et al. PetaBricks: A language and compiler for algorithmic choice In *Proc. PLDI*, 2009.
[4] P. Balaprakash et al. Can Search algorithms save large-scale automatic performance tuning? Technical report ANL/MCS-P1823-0111, Argonne National Laboratory, January 2011.
[5] R. R. Barton and J. S. Ivey, Jr. Modifications of the Nelder-Mead Simplex method for stochastic simulation response optimization. In *Proc. WSC*, 1991.
[6] C. Chen et al. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *Proc. CGO*, 2005.
[7] K. A. De Jong. *An analysis of the behavior of a class of genetic adaptive systems.* PhD thesis, Ann Arbor, MI, USA, 1975.
[8] B. Efron. R.J. Tibshirani. *An introduction to the bootstrap.* New York: Chapman & Hall, 1993.
[9] M. Frigo and S. Johnson. Fftw: an adaptive software architecture for the FFT. In *Proc. IEEE ICASSP*, volume 3, pages 1381–1384, 1998.
[10] T. Katagiri et al. Fiber: A generalized framework for auto-tuning software. In *Proc. ISHPC*, 2003.
[11] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proc. IEEE Int. Conf. on Neural Networks*, Piscataway, NJ, 1995.
[12] Z. Michalewicz and D.B. Fogel. How to Solve It: Modern Heuristics. Springer Verlag, 2004.
[13] M. Puschel et al. Spiral: code generation for dsp transforms. *Proc. of the IEEE*, 93(2), 2005.
[14] I. Rechenberg. *Evolutionsstrategie: Optimierung Technischer Systeme nach Prinzipien der biologischen Evolution*. Frommann-Holzboog, 1973.
[15] R. Ribler et al. Autopilot: Adaptive control of distributed applications. In *Proc. IEEE HPDC*, 1998.
[16] K. Seymour et al. A Comparison of search heuristics for empirical code optimization. In *Proc. CGO*, 2008.
[17] M. Stephenson et al. Meta optimization: improving compiler heuristics with machine learning. In *Proc. PLDI*, 2003.
[18] R. Storn and K. Price. Differential evolution- a simple and efficient adaptive scheme for global optimization over continuous spaces. Technical report, 1995.
[19] C. Tapus et al. Active harmony: Towards automated performance tuning. In *Proc. HPNC*, 2003.
[20] A. Tiwari et al. A scalable auto-tuning framework for compiler optimization. In *Proc. IPDPS*, pages 1–12, 2009.
[21] R. Vuduc et al. Oski: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16(1):521+, 2005.
[22] Z. Wang and M. F. O'Boyle. Mapping parallelism to multi-cores: a machine learning based approach. In *Proc. PPoPP*, 2009.
[23] C. R. Whaley et al. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1-2):3–35, January 2001.
[24] K. Yotov et al. Is search really necessary to generate high-performance blas? *Proc. of the IEEE*, 93(2):358–386, February 2005.