Karlsruher Institut für Technologie

Karlsruhe Reports in Informatics 2010,8
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

Search Algorithms for Automatic Performance Tuning of Parallel Applications on Multicore Platforms

Victor Pankratius

2010

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

Search Algorithms for Automatic Performance Tuning of Parallel Applications on Multicore Platforms

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

ABSTRACT

Multicore processors bring parallelism to every desktop computer, but multicore application tuning can be a costly challenge because of the increasing diversity of parallel platforms. Hard-coded program optimizations for one platform might not work on others, which harms program portability. Auto-tuners can tackle this problem by re-tuning applications on every new platform prior to productive execution; they search for optimum performance by using run-time feedback information and adjusting the performance-critical parameters of a program in a loop. An important question that has not been answered satisfactorily so far is which tuning algorithms work well for multicore applications on today's desktops and servers, and how effective they are. This paper provides quantitative answers for a range of algorithms that don't require application-, input-, or platform-specific performance models. It proposes a novel approach of binary search sampling combined with non-linear model prediction, which requires fewer iterations and has an error an order of magnitude lower than other classical optimization approaches. The empirical analysis is based on data from more than 60 experiments with 113 workloads, gathered from 22 multicore applications on 9 multicore platforms. The data also provides evidence that most applications perform better with a non-intuitive number of threads; this number is often significantly higher than the number of hardware threads and varies among platforms.

1. INTRODUCTION

With multicore chips, parallel computing has arrived on every desktop. The parallelization of a large number of performance-critical applications is a great challenge, but it's not the only one. An additional problem is parallel performance optimization. Present day multicore platforms differ in many hardware and software characteristics, such as the number of cores, threads per core, clock rate, cache architecture, memory bandwidth, or operating system – to name just a few. This complicates performance tuning because a code optimization targeted at one platform might lead to bad results on others. The problem is serious, as it might offset all benefits of parallelization. Hand-tuning as a last resort confronts on the one hand programmers with tedious tasks and on the other hand managers with exploding costs for portability maintenance, due to combinatorial state-space explosion of platform configurations.

Auto-tuners provide a solution to this problem by automating for a program the empirical search of the software parameter value configuration that leads to the best performance on a particular platform. If multicore applications are made tunable (e.g., via command line parameters), they can be iteratively re-tuned after migration to other platforms, based on run-time feedback information gathered on a particular platform. From the software engineering point of view, auto-tuning separates performance tuning concerns from code development concerns and simplifies portability. Tedious performance experiments can be delegated to an auto-tuner. Programmers have more time to focus on parallel code that is less complex, easier to read, understand, and maintain. These key issues help reducing part of the complexity and cost of parallel programming.

Which algorithms should an auto-tuner use for tuning? How well do they work and how do they compare to each other? This question has not been answered satisfactorily so far for parallel applications running on today's multicore desktops and servers. This paper will focus on tuning algorithms that do not require specific performance models for each application, input, and platform, and shows that it is nevertheless possible to obtain good results with algorithm-implicit general models. This approach is very beneficial because it avoids dealing with state-space explosion of platform configuration information. We contrast tuning algorithms inspired by classical optimization, sampling, and non-linear modeling, and present own approaches.

We validate the proposed algorithms using a total of 22 multithreaded applications taken from different areas, such as project management, virus scanning, numerics, and graphics. Several of these applications were developed by different teams over a period of several months. We conducted over 60 experiments on 9 multicore platforms covering a spectrum of hardware and software characteristics. The results show that our approach based on binary search sampling combined with non-linear curve fitting prediction works well; compared to classical optimization based on hill climbing or simulated annealing, we need fewer iterations and obtain errors by an order of magnitude lower. In addition, we provide empirical evidence falsifying common belief that the maximum number of software threads must equal the number of hardware threads. In fact, most programs perform better with significantly more software threads than hardware threads, indicating that latency-hiding is useful.

The paper is organized as follows. Section 2 presents the motivation, principles, and concepts of auto-tuning. Section 3 discusses auto-tuning algorithms. An experimental evaluation is done in Sections 4 and 5; Section 4 details the experimental setup, and Section 5 the results. Section 6 shows

Technical Report 2010-8
Karlsruhe Institute of Technology (KIT), Germany
Institute for Program Structures and Data Organization (IPD)
19 July 2010

how software engineers can put the insights of this paper into practice by using Autotunium, an extensible platform-independent tuner. Section 7 contrasts related work, and the conclusion in Section 8 summarizes key insights.

2. AUTO-TUNING – AN OVERVIEW

This Section presents the motivation, the general principles, and the definition of auto-tuning used in this paper.

2.1 Is Auto-Tuning Necessary?

Figure 1 shows the performance of two real multithreaded applications. The left table shows performance results from PBzip2, an open-source multithreaded compression program; the right table presents a multithreaded project management application estimating project durations stochastically. Both applications (described in Section 4.2) can be configured to run with a different number of threads and are executed with the same respective inputs on different multicore platforms (see Table 1). The code of the second application has also been compiled with two different compilers.

PBzip2

Platform	7	8	9
#Threads at max speedup	9	34	60
Max speedup	6,2	13,1	10,0
Potential performance penalty			
Fixing #threads at 8	1%	79%	68%
Fixing #threads at 16	3%	27%	29%
Fixing #threads at 32	13%	3%	7%
Fixing #threads at 64	14%	4%	1%

Parallel Project Management App.
(Compiler: VS:Visual Studio, I: Intel)

2-VS	2-I	3-VS	3-I	6-VS	6-I
39	54	60	63	62	8
2,0	2,0	1,9	1,9	5,8	6,2
Potential performance penalty					
5%	31%	12%	22%	94%	0%
2%	1%	12%	19%	45%	52%
1%	3%	5%	2%	13%	94%
1%	2%	5%	1%	7%	111%

Figure 1: Performance penalty maps illustrating a migration scenario for two multithreaded applications.

In general, many programmers tend to fix the maximum number of threads to some constant (e.g., the number of hardware threads). Is this a good idea? A migration scenario for our two applications shows that this intuitive approach can be inefficient. To be able to compare, we exhaustively tried out all thread counts from 2 to 64 on each platform and determined the speedup maximum. The penalty maps illustrate what would happen if the number of threads is fixed while the application is migrated to another platform: None of the intuitively chosen thread counts works equally well on all platforms! In addition, the optimum thread count also depends on other factors, such as the chosen compiler. Using intuitive thread numbers can lead to significant slowdowns on other platforms (79% and 111% in the worst case). This case study example illustrates with real empirical data that hard-coding performance parameters such as the number of threads is not a good idea. It can be difficult to intuitively guess an optimum value that remains applicable after program migration, so an context-based adaptation is necessary. The results motivate this paper to focus the empirical evaluation on the number of threads as one of the most important types of performance parameter, even though most of the presented algorithms can be applied to other types of metric-scaled, numerical parameters.

2.2 Auto-Tuning Principles

This paper assumes that auto-tuning is a discrete optimization problem defined in the following way: Starting point is a parallel program with input I and z tunable parameters $a = \{a_1, \ldots, a_z\}$. We assume that the program has been developed in a way that an auto-tuner can set values for each a_i $(1 \leq i \leq z)$, which are chosen from a predefied domain $dom(a_i)$. Examples for such parameters are: The number of threads in an application thread pool, the maximum number of workers in a master-worker pattern, the block size in KB of a particular buffer (which can affect cache behavior), etc. The entire configuration search space is defined by $C = dom(a_1) \times dom(a_2) \times \ldots \times dom(a_n)$. The auto-tuner searches for a configuration $c \in C$ that maximizes a given objective function f (which is typically the speedup or the negative execution time of the program). To evaluate $f(c)$, the auto-tuner executes the program with input I and each parameter a_i set to value c_i and gathers feedback information used to compute a new vector c. The process repeats in a loop until some termination criterion holds.

Offline auto-tuners compute new tuning parameter configurations between different program runs. For example, feedback information can represent the entire execution time of a program or the execution time of a particular function instrumented with measurement probes. Generating configurations for new runs can be realized for example by setting command line parameters or by recompiling the whole program after code transformation or generation. By contrast, online tuners work during program run-time and apply a variety of techniques (e.g., machine learning [4]) to tune long-running programs which execute some functionality in a repeated fashion (e.g., multimedia decompression on video streams). This work concentrates on offline tuning. We assume that the same program is executed with the same input several times. The paper investigates empirical search algorithms how to traverse the search space, obtain new configurations $c \in C$ in each tuning run, and how to find the best speedup for an entire application.

3. EMPIRICAL SEARCH ALGORITHMS

This Section presents auto-tuning algorithms for empirical search that are inspired from different fields, such as combinatorial optimization, sampling, and nonlinear modeling. We start with a discussion of general requirements.

3.1 Requirements

Auto-tuning search algorithms should deliver high-quality configurations $c \in C$ within an acceptable time frame.

Given c_{max} as the configuration of the true global optimum and c_{found} as the best solution found by an algorithm, result quality can be measured by the difference $f(c_{max}) - f(c_{found})$ and by the relative error $(f(c_{max}) - f(c_{found}))/f(c_{max})$. The problem is that c_{max} is usually unknown, and that the completeness property that the global optimum is found can only be guaranteed either by searching the entire space or by having additional information that can be used to safely prune configurations and exhaustively search the remaining space. Either way, exhaustive search can be impractical; it is therefore reasonable if a search algorithm has the *Probabilistic Approximately Complete (PAC)* property, which is a convergence criterion meaning that given enough iterations, the probability of not finding the optimum solution is arbitrarily small [9].

An additional constraint of practical relevance is that each algorithm iteration requires a program execution that may take a long time. An existing optimum should be found with as few iterations as possible. This constraint influences the design and choice of auto-tuning algorithms and differs from many classical optimization problems that assume an infinite or arbitrary large number of iterations.

In the following Sections, all algorithms will share some characteristics. The configuration with the lowest execution time also has the highest speedup, so we use speedup to better compare results from different platforms. Every algorithm

has the input *data* that represents the search space by a set of pairs $data = \{(x_2, y_2), \ldots, (x_n, y_n)\}$, with n as the maximum number of threads. For each pair $p_i = (x_i, y_i)$ with $i, n \in \mathbb{N}$ and $2 \leq i \leq n$, $y_i \in \mathbb{R}$ represents the speedup that was measured with x_i threads. Every algorithm computes a pair $p \in data$ with the highest speedup found.

3.2 Local Search Approaches

Local search techniques are widely applied for optimization. They require a starting point in the search space, and iteratively move to neighboring points – based on local knowledge only – until some termination criterion is satisfied. The search trajectory through the search space consists of a finite sequence of visited points. Two classical combinatorial optimization algorithms are analyzed in the context of auto-tuning: Hill Climbing (Algorithm 1) and Simulated Annealing [11] (Algorithm 2).

Algorithm 1. Auto-Tuning based on Hill Climbing

Input: $start \in data$ **Output:** $p \in data$
1: $ready \leftarrow false$; $visited \leftarrow \emptyset$; $current \leftarrow start$
2: **while** not ready **do**
3: $visited \leftarrow visited \cup current$
4: $next \leftarrow$ pick point with highest speedup from {current ∪ direct neighbors of current}
5: **if** current==next **then** $ready \leftarrow true$
6: **else** $current \leftarrow next$
7: **end if**
8: **end while**
9: **return** $p \in visited$ that has highest speedup

Hill Climbing iteratively improves a starting solution, but can get stuck in a local optimum. Simulated Annealing [11, 14] was proposed as an analogy to thermodynamic phenomena; in our context, it can overcome a local optimum as follows: if the speedup $S(next)$ of a newly picked point is worse than the speedup of the current point $S(current)$, then the algorithm can move to the new point with the probability $e^{\frac{S(next)-S(current)}{temp}}$. The parameter *temp* is changed in each run according to a so-called cooling function. The basic idea is that more random moves are allowed in early iterations – potentially escaping local optima– while in later iterations the algorithm behaves more and more like hill climbing.

Algorithm 2. Tuning based on Simulated Annealing

Input: $start \in data$; $maxIters \in \mathbb{N}$; $temp \in \mathbb{R} > 0$
Output: $p \in data$
1: $ready \leftarrow false$; $visited \leftarrow \emptyset$; $current \leftarrow start$
2: **for** $i = 0$; $i < maxIters$; i++ **do**
3: $visited \leftarrow visited \cup current$
4: $next \leftarrow$ random direct neighbor of current
5: **if** S(current)<S(next) **then**
6: $current \leftarrow next$
7: **else**
8: **if** $random[0,1) < e^{\frac{S(next)-S(current)}{temp}}$ **then**
9: $current \leftarrow next$
10: **end if**
11: **end if**
12: $temp \leftarrow coolingFunction(temp)$
13: **end for**
14: **return** $p \in visited$ that has highest speedup

3.2.1 Properties and Questions

The local knowledge used to make decisions in both of the algorithms is incomplete and the same location may be visited several times. Only Simulated Annealing has the *PAC* property guaranteeing convergence to the global optimum, but only if the number of iterations is sufficiently large (potentially infinite) [11, 14].

Several questions need to be answered empirically later in the paper. How well do these algorithms work for finding the optimum speedup of multicore applications? Are there many local optima in which the algorithms get stuck or is this problem less relevant, e.g., because there are just a few optima close to the global optimum? How many iterations are typically needed? In addition, no general recommendations for initial temperature or cooling function are available; they are both context-dependent and usually determined in an experimental way.

3.3 Sampling Approaches

Sampling techniques are appealing for tuning because they can be designed to work in a simple way, i.e., with potentially less overhead than other techniques. We investigate non-adaptive techniques and also propose a novel adaptive technique, called binary search sampling, which steers the number of samples as well as the region from which samples are taken based on an implicit model.

3.3.1 Random Sampling

Picking $1 \leq j \leq n$ random samples $p_j \in data$ can be done in two ways: With replacement **(Algorithm 3)** which means that a particular value p_j can be sampled more than one time, or without replacement **(Algorithm 4)** where a particular p_j can be picked exactly one time.

For a given sample size j, it is generally known that sampling with replacement is inherently less efficient than sampling without replacement, because the variance of the sample mean is lower [24]. From the tuning perspective, however, sampling with replacement can be advantageous for programs whose executions with the same parameter need not be repeated one more time, and if the speedup value can be retrieved from a temporary cache populated with values from earlier runs. The experimental evaluations presented later will quantify the error difference between tuning based on sampling with and without replacement.

3.3.2 N-Step Sampling

This is a systematic sampling technique, which arranges a population in a certain order and samples according to a regular pattern [24]. N-step sampling assumes that all pairs $(x_2, y_2), \ldots, (x_n, y_n)$ are ordered by x_i and picks every N-th pair **(Algorithm 5)**. Programmers often use this pattern to tune the number of threads when they cannot afford to exhaustively sample the entire parameter space.

3.3.3 Binary Search Sampling

We introduce binary search sampling as a novel adaptive sampling method for performance tuning (Algorithm 6). In contrast to random sampling, it guides the sampling process and takes advantage of population characteristics to sample more interesting values with a higher probability. The purpose is to gain precision (in finding high speedups) and efficiency (i.e., with fewer iterations). It assumes an implicit model in which a locality principle holds, i.e., if it finds a point with good speedup, it assumes that other points in an interval around that point have good speedups, too.

In principle, binary search sampling works as follows: The search space is divided into two halves. Random samples are

taken from each half. After determining which half contains the sample with the highest speedup, the algorithm repeats the procedure for that half. If both halves contain a point with equal maximum speedup, one half is chosen at random. This is done as long as the width of the sampling interval is greater than the number of samples that need to be taken.

Algorithm 6. Auto-Tuning based on Binary Search Sampling with Fixed Sample Size #s

Input: $\#s \in \mathbb{N}$; $visited \leftarrow \emptyset$, $min = 2$, $max = n$
Output: $p \in data$
1: **while** $(max - min) > \#s$ **do**
2: $mid = \lfloor min + (max - min)/2 \rfloor$
3: $left = \{s_1, \ldots, s_{\#s} | s_i$ rnd from $[p_{min}, p_{mid}[\ \}$
4: $maxL \leftarrow y_i \mid (x_i, y_i) \in left$ and $\nexists (x_j, y_j) \mid y_j > y_i$
5: $right = \{s_1, \ldots, s_{\#s} | s_i$ rnd from $[p_{mid}, p_{max}]\ \}$
6: $maxR \leftarrow y_i \mid (x_i, y_i) \in right$ and $\nexists (x_j, y_j) \mid y_j > y_i$
7: $visited \leftarrow left \cup right$
8: **if** $maxL < maxR$ **then** $min = mid$
9: **else if** $maxL > maxR$ **then** $max = mid$
10: **else if** $randomchoice(\{0, 1\}) = 0$ **then** $min = mid$ else $max = mid$
11: **end if**
12: **end while**
13: **return** $p \in visited$ that has highest speedup

We propose two versions: With fixed sample size (**Algorithm 6**) and with dynamic sample size (**Algorithm 7**). With a fixed sample size, the algorithm takes from every half a predefined constant number of samples. In the dynamic version, the number of samples to take from each interval is specified by a predefined percentage of the interval width. This means that more samples are taken during early iterations, while the number of samples decreases later on; the rationale is that this strategy avoids picking the wrong bigger interval, so we don't search for good speedups in a totally wrong place. We omit the pseudocode for the dynamic version (Algorithm 7) as it contains just minor modifications. The probability of picking a particular thread-speedup pair depends on which pairs were sampled earlier, so the probability of being picked is conditional (samples are not taken from discarded intervals).

Sampling the interval with the *rnd* function (in Algorithm 6) can be done with and without replacement. In the version with replacement, one particular sampled point can be sampled again (though it would not count as a new iteration if the old value is cached). Without replacement, the algorithm will never re-sample a point that was already sampled. It chooses another point if there are free ones in the current interval; if all available points were sampled earlier, *rnd* returns the empty set. We did experiments with and without replacement, but present later only the better results, which were obtained for sampling without replacement. We also remark that the total number of samples can vary: In some cases, the best values might have been already sampled in early iterations. When the intervals become iteratively smaller, the algorithm has already encircled good values and might not find more value to sample. In other situations, such small intervals might have more points available (that were not sampled in earlier iterations).

3.3.4 Properties and Questions

Sampling algorithms have the advantage that the number of tuning iterations can be predicted, and more iterations lead to better predictions. The probabilistic behavior helps escape local optima. The algorithms obviously have the *PAC* property – with a large number of executions they will eventually visit all points in the search space. Question is whether it is possible to guide the sampling process towards better solutions and thus reduce the number of iterations.

Random sampling has no mechanism for steering the sampling process. Nevertheless, it can be used as a benchmark for tuning quality: A tuning algorithm requiring the same number of samples $p \in data$ can be compared to random sampling and should produce a lower error to justify the additional implementation effort. Random sampling can be used as a benchmark to compare tuning approaches and contrast results in the literature.

N-step sampling is vulnerable to periodicities in an ordered population. Using the right/wrong sampling interval can lead to results better/worse than random sampling, which most programmers are unaware of. The paper will present experimental results for good values of N for the workloads introduced later.

Binary search sampling guides the sampling towards potentially favorable locations, and the number of steps is asymptotically logarithmic in the search space size. Experiments will analyze the effectiveness and quantify the error depending on the number of samples.

3.4 Prediction with Non-linear Models

This approach works as follows (**Algorithm 8**): First, gather k pairs $p = p_1, \ldots, p_k \in data$. Given a function $speedup = f(numthreads)$, find the parameters of f that fit f best on p, i.e., which minimize the sum of square errors. Then, the maximum of f is determined using calculus, i.e., we obtain a prediction at which number of threads t the speedup curve reaches its highest value. As curve fitting works with non-discrete functions, the result is mapped to the pair $p_j = (x_j, y_j)$ whose x_j is closest to t. Finally, the algorithm outputs the pair with the highest speedup from all visited points $p_j \cup p$.

The overall goal is to make accurate speedup predictions for the entire population *data* with a low number k of pairs. Pairs used for curve fitting can be gathered in different ways, e.g., using the sampling techniques described in Section 3.3.

A key question is which model to use for the prediction function f. An intuitive choice could be a higher-order polynomial, based on the assumption that higher orders can better approximate the data. However, we hypothesize for multicore workloads that other models work better in practice than high-order polynomials, for the reason illustrated in Figure 2.

The Figure shows that an 8th degree polynomial has a lower error (i.e, higher R^2) when fitting the samples, but introduces unrealistic swings that misguide the prediction for the actual maximum speedup. The lower-order polynomial's error is slightly higher, but the curve averages the results in a more favorable way for the overall speedup prediction. This observation suggests that the shape of lower order polynomials needs to be corrected using other functions, instead of increasing the order of the polynomial.

We consider the following functions to include in the non-linear model: (1) $Log(x)$ can model a typical workload shape for parallel applications that improve speedups with an increasing number of threads, but where increases at higher thread numbers are lower due to overhead or Amdahl's law. (2) $Tanh(x)$: The hyperbolic tangent has a sigmoid shape that is useful for modeling speedup increases and decreases that may be caused by saturation effects or communication. (3) $Sin(x)$ and $Cos(x)$ may better capture swings and peri-

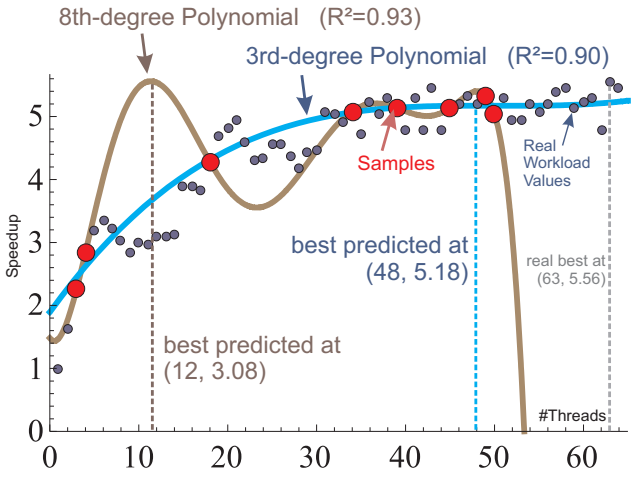

Figure 2: Example of a prediction with two non-linear models.

odicity in the workload shape.

In summary, the following model alternatives will be evaluated and compared experimentally:

$$f_1\text{: } y = a + b \cdot x + c \cdot (x+d)^2$$

$$f_2\text{: } y = a + b \cdot x + c \cdot (x+d)^2 + e \cdot (x+f)^3 + g \cdot Log(x) + h \cdot Tanh(i \cdot x + j)$$

$$f_3\text{: } y = a + b \cdot x + c \cdot (x+d)^2 + e \cdot (x+f)^3 + g \cdot Log(x) + h \cdot Tanh(i \cdot x + j) + k \cdot Sin(l \cdot x + m) + n \cdot Cos(o \cdot x + p)$$

$$f_4\text{: } y = a + b \cdot x + c \cdot (x+d)^2 + \ldots + o \cdot (x+p)^8$$

The simple model f_1 assumes that there is a sweet spot for parallelization where the optimum speedup is reached, i.e., too few threads do not exploit all parallelization potential, whereas too many threads cause overhead and slowdowns. Model f_2 aims to improve the prediction accuracy of f_1 by accounting for typical workload shapes. Model f_3 aims to improve f_2 by accouting for swings and periodicity. Model f_4 is an 8th degree polynomial used for comparison.

3.4.1 *Properties and Questions*

Compared to random sampling, non-linear curve fitting introduces a model to improve the prediction accuracy. As it's based on sampling, it also has the *PAC* property. Curve fitting requires in the worst case an additional iteration and more complex computations. The question which model works best will be answered experimentally; even though f_3 and f_4 might be able to model more shape variations, they also need more samples and might lead to over-fitting. Another question to be answered is how to sample the data used for curve fitting; it will be shown that a combination with binary search sampling works better than random sampling.

4. EXPERIMENTAL SETUP

We describe next the experimental approach, the employed benchmark applications, and the workloads.

4.1 Methodology

We evaluate the proposed algorithms on 113 real workloads. First of all, the real workloads are collected by executing each application exhaustively with $1 \ldots 64$ threads in increments of 1 on various multicore platforms. We compute for each thread number the average execution time of several runs to avoid noise, and the speedup in relation to the execution time with 1 thread.

To be able to compare results, we use a controlled environment in which we simulate how an auto-tuner would work on a real program, except that instead of executing the program, we look up the data in the existing workloads. In addition, we exhaustively search for the global speedup optimum in each workload, which allows us to compute the error between a particular algorithm output and the real optimum. This is one of the first papers to do so in a systematic way, using many realistic multicore workloads.

We try out every algorithm 1000 times on each of the 113 workloads and collect in every trial the number of required iterations and the relative error. The number of trials has been determined experimentally as it has not shown significant convergence improvements for higher numbers (e.g., 10,000 and 100,000 trials). If the same point in a workload is visited more than one time, we count it as one iteration; this is because we assume that in reality the measurement would also not be done twice, but looked up in a cache. We finally generate empirical distributions of the average error and standard deviation for a particular number of iterations, as well as a distribution of the number of iterations.

The complete experimental environment as well as all algorithms have been implemented in Mathematica [27]. This was beneficial for the symbolic computations needed to solve the optimization problem in the non-linear curve fitting approach (the concrete model functions differ for every sample set, workload, and trial).

4.2 Benchmark Applications

We use a total of 22 multithreaded applications that run on multicore platforms. For some of the applications, several versions were implemented using different parallelization strategies, I/O strategies, libraries, languages, and compilers. In addition, most of the applications were developed by different teams over a period of several months. All applications are tunable, i.e., they are written in a way that tuning parameters (e.g., number of threads and others) can be set via command line parameters. The applications are summarized as follows:

Five parallel compression applications. Four different multithreaded versions of the BZip2 algorithm were implemented by different developers using C++, Pthreads, and gcc (three versions were part of a lab project [18]). In addition, we use pbzip2 [8], an open-source multithreaded implementation written in C++ and Pthreads. Bzip2 works on independent data blocks passing a pipeline of algorithms. The applications differ in the employed parallelization strategies and in the way they employ task parallelism, data parallelism, and parallel patterns such as master-worker or producer-consumer. Implementation details are shown in [18].

Three parallel virus scanners. The sequential, open-source ClamAV virus scanner was parallelized by three different teams during a semester project, using C++, Pthreads, and gcc. In principle, all teams have a queue or some other data structure filled with work units (i.e., lists of files) that are taken and processed by several threads. Each thread independently scans the files of a work unit for virus signatures (we used fake signatures in our tests, not real viruses). The team's implementations differ in the granularity of work within a work unit and in the swapping strategies from disk to main memory.

Eight parallel project management applications. These are variant implementations in C++ with OpenMP

of a project management application for stochastic project duration prediction. It uses Monte Carlo simulation to estimate the overall project duration based on probability distributions of individual project tasks. Four of the applications perform I/O and continuously write temporary results to disk, whereas the other four perform reductions in main memory and just write the final histogram to disk. The implementations also differ in the strategies and employed constructs that synchronize histogram updates. Each application was compiled with the Visual Studio C++ compiler and with Intel's C++ compiler. Details are sown in [19].

Four parallel matrix multiply programs. Matrix multiply is an important numerical kernel in many compute-intensive applications. It is not only important for scientific computations, but can also be applied to other areas, such as finding shortest paths in graphs [1]. In contrast to the other applications, it contains most of the work in loop structures. We use two Java implementations and two C++ with OpenMP implementations in gcc [13]; the implementations differ in the way the loops iterate over matrices, which influences cache usage, offering potential for superlinear speedups.

One parallel Mandelbrot Set calculator. We use this application as a representative for compute-intensive, embarassingly parallel computations. It is written in Java and generates an RGB image [13].

One parallel ray tracer. The open-source Tachyon ray tracer [23] is used as an example for a parallel graphics application. It is written in C++ and Phthreads.

4.3 Workloads

Due to the combinatorial explosion, it would take a long time to exhaustively benchmark all applications with 1 to 64 threads on all of our multicore platforms and operating systems (see Table 1). The selected workloads that form our total population try to distribute various characteristics in several dimensions: Out of 113 workloads, about half perform file I/O (49 perform file I/O, 64 do not). The number of workloads on each hardware platform is distributed as follows: 28 are executed on Intel 2-core, 12 on Intel Quadcore, 34 on Intel 8-core, 17 on Niagara1 8-core, and 22 on Niagara2 8-core platforms. We have more workloads from 8-core platforms because they become widely spread, but on the other hand we do not want to completely ignore platforms with fewer cores. The workload distribution for operating systems is as follows: 46 on Windows OS, 28 on Linux, and 39 on Solaris.

As we test all algorithms on one workload (and this for many workloads), we do not require workloads to be evenly distributed across machines. Parallel compression has 26 workloads, all with an input file of about 80 MB gathered from machines 7, 8, and 9. Parallel virus scanning has 3 workloads obtained from machine 8, using an input directory containing over 7000 files of about 300 MB in total. Parallel matrix multiply has 24 workloads gathered from machines 5, 7, and 9; as inputs, we used square matrices of size 512, 1024, and 2048. The parallel project management application has 42 workloads gathered from machines 2, 3, and 6; as described earlier, we employed 8 parallelization strategies and two different compilers. All workloads use the same input file, which is a project schedule with 17 tasks. Parallel Mandelbrot has 12 workloads gathered from machines 4, 8, and 9, and uses square image output sizes of 1024, 2048, 4096, and 8192 pixels. Parallel ray tracing has 6 workloads gathered from machines 5, 7, and 9; it uses a 389KB and 453KB scene description file as input and generates 669x834 and 512x512 pixel TGA images that are written to disk in half of the cases, and just displayed in all other cases.

5. EXPERIMENTAL RESULTS

This section first sketches some descriptive statistics of our multicore workloads. We then present the experimental algorithm evaluations and discuss potential threats to validity.

5.1 Workload Descriptive Statistics

Figure 3 presents a performance overview for all 113 workloads. The x-axis groups workloads by the platform on which they are executed. The empirical workload data shows an interesting non-intuitive result: Most applications have the best speedup at a number of software threads that is higher than the number of hardware threads (see Table 1), possibly due to latency hiding with more threads. Some programs have superlinear speedups due to cache effects. The results contradict popular recommendations to set the number of software threads to the number of hardware threads. In addition, most optimum thread counts are not a power of 2, which applies to 107 (95%) of the workloads. Out of 113 workloads, 87 (77%) have the best performance at a thread count greater than 8 and 58 (51%) at a thread count greater than 39. Splitting up specific application workloads, the best performance with a thread count greater than 8 applies to 88% of compression workloads, 100% of Mandelbrot workloads, 33% of matrix multiply workloads, 34% of project management workloads, 33% of ray tracing workloads, and 100% of virus scan workloads.

5.2 Algorithm Evaluations

Figure 4 shows aggregated results for all algorithms described in Section 3. It also shows for different algorithm parameters the resulting median number of iterations and the average error. Additional details on the empirical distributions are illustrated in Figure 5.

(1) Hill Climbing has the worst result of all algorithms. Different choices for the starting point (e.g., with 2 threads, random, or a random power of 2) all produce higher errors than most of the other algorithms. The first variant requiring 7 iterations is even worse than random sampling with 5 samples. Choosing the starting point at a random power of 2 is slightly better than the other choices. The explanation for the bad results is that most workloads are not smooth and have many local maxima in which this algorithm gets stuck. In addition, Figure 5 (a) shows a large error standard deviation and a large variance in the number of iterations. More iterations do not reduce the error.

(2) Simulated Annealing is evaluated with a cutoff of 20 and 40 iterations as well as different cooling functions commonly suggested in the literature [14]. Except for one configuration in which the cooling function cools off faster than the others, the average error improves in comparison to hill climbing, but is still worse than most other algorithms. Figure 5 (b) reveals large error and iteration variances, but in contrast to hill climbing, more iterations reduce the error.

(3) Random Sampling and (4) N-Step Sampling. Surprisingly, random sampling with and without replacement has better tuning results than the classical optimization algorithms (1) and (2), and even requires fewer samples. Random sampling sets the bar that other tuning algorithms must beat. As expected, sampling without replacement has slightly lower errors than sampling with replacement [24]. Figure 5 (c) quantifies the difference, which becomes bigger at a higher number of samples. The results for N-step sampling show for step sizes between 2–4 or 6–8 lower errors than sampling without replacement, possibly due to exploitation of periodities. At the same time, N-step sampling does not perform well with an inappropriate step size.

Machine	Processor(s)	Clock[GHz]	Cores	HWThreads/Core	L2Cache[MB]	RAM[GB]	OS
1	Intel E6600	2.4	2	1	4	2	WinVistax32
2	Intel T2500	2	2	1	2	2	WinXPx32
3	Intel E6400	2,13	2	1	2	3	WinVistax32
4	Intel Q6600	2,4	4	1	8	4	Win7x64
5	Intel Q6600	2,4	4	1	8	4	Ubuntu Linux8x32
6	2xIntel5320QC	1.86	2x4	1	8	8	Win2003x64
7	2xIntel5320QC	1.86	2x4	1	8	8	Ubuntu Linux8x32
8	Sun Niagara 1 T2000	1	8	4	3	16	Solaris5.1
9	Sun Niagara 2 T5120	1,2	8	8	4	16	Solaris5.1

Table 1: Multicore platforms used for benchmarking

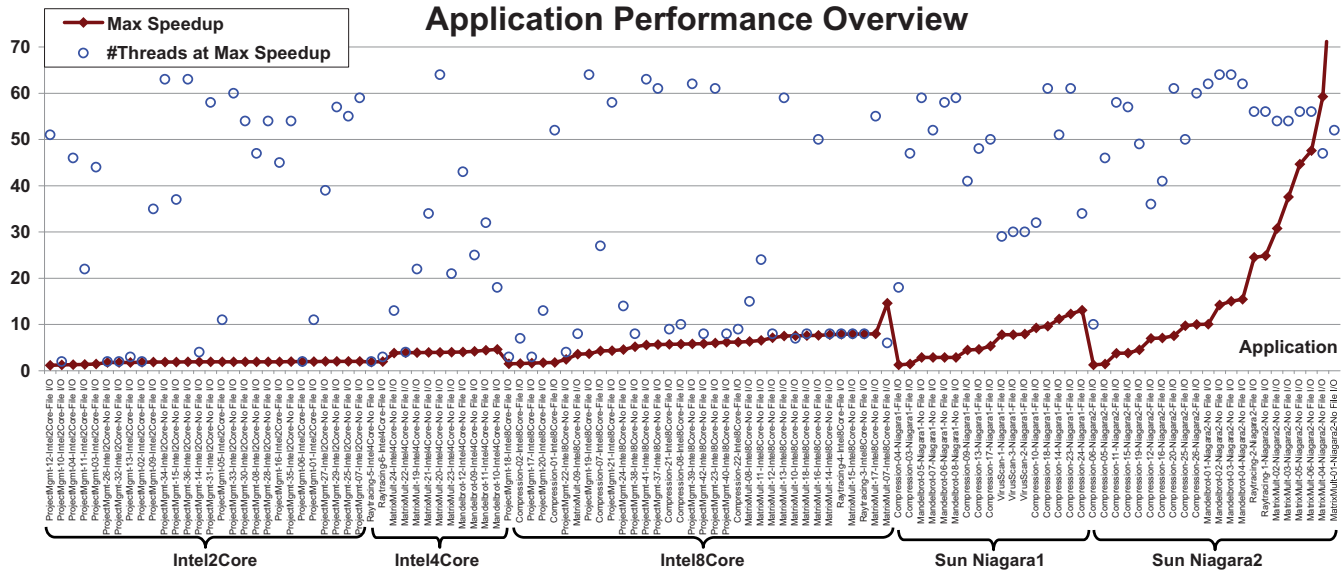

Figure 3: Performance Overview

(5) Binary Search Sampling beats all algorithms discussed so far in terms of average error. This observation applies to both variants (with fixed and with dynamic samples size). Thus, the experiments support the hypotheses of the underlying model that guides the sampling mechanism. The approach with a fixed number of samples per interval has lower errors than the dynamic approach that adapts the number of samples to the interval length. This result implies that it is less efficient to have more samples in early iterations and fewer samples in later iterations, as the algorithm might miss the relevant points when it finally gets to a smaller promising region. The empirical distributions in Figure 5 (d) are more stable and better predictable, compared to hill climbing and simulated annealing. We remark that we also conducted experiments for binary search sampling with replacement. We omit, however, their presentation and present just the best results. The results of binary search sampling with replacement are slightly worse compared to the versions without replacement.

(6) Prediction with Non-Linear Models. Figure 4 shows that model f_1 does not lead to significant improvements over random sampling without replacement, which suggests that this model does not capture all relevant aspects. By contrast, the model f_2 has consistently better results than all other models. For the employed workloads, it supports our hypothesis that models other than high-order polynomials can have a better prediction accuracy, and supports the assumptions behind the functional components of f_2. Models f_3 and f_4 are better than random sampling, but worse than model f_2. All curve-fitting approaches, however, are worse than binary search sampling with a similar number of iterations.

(7) Binary Search Sampling combined with Non-Linear Modeling. As binary search sampling produces better results than random sampling, the results of curve fitting can be improved if the samples used to fit the model are chosen differently. Figure 4 and Figure 5 (e) show that the strategy combining the best binary search sampling strategy (fixed number of samples, without replacement) with the best curve fitting model (f_2) works well and produces the lowest errors. The Figure also illustrates that the advantage of the combination can be attributed to the lower errors made for a smaller number of iterations. The advantage becomes smaller when the number of iterations increases.

5.3 Threats to Validity

As described in Section 4, all experiments are done in a controlled environment on workloads generated exhaustively from real programs. Due to the combinatorial explosion, however, it is difficult to exhaustively benchmark all applications on all platforms with all parameter permutations. A selection is necessary, which can influence the results. We are confident, however, that the criteria and the aggregated workload properties described earlier offer an acceptable tradeoff. As we compare all algorithms on one workload (an repeat this process for many different workloads), we do not create workloads for every permutation of parameter values and appli-

	Method	Parameters	#Iter	AvgError
1	Hill Climbing(1)	startIdx: 2	7	18,92%
2	Hill Climbing(2)	random	2	18,18%
3	Hill Climbing(3)	2^random	1	16,24%
4	SimAnnealing(1)	maxiters: 20, CoolF: T=0.9T, Tinit=20	7	12,59%
5	SimAnnealing(2)	maxiters: 40, CoolF: T=0.9T, Tinit=20	9	11,36%
6	SimAnnealing(3)	maxiters: 20, CoolF: T=0.9T, Tinit=10	6	11,67%
7	SimAnnealing(4)	maxiters: 20, CoolF: T=0.5T, Tinit=20	5	19,46%
8	Random withRepl(1)	5 samples	5	6,72%
9	Random withRepl(2)	10 samples	10	4,45%
10	Random withRepl(3)	15 samples	15	3,42%
11	Random withRepl(4)	20 samples	20	2,78%
12	Random withRepl(5)	25 samples	25	2,34%
13	Random withRepl(6)	30 samples	30	2,00%
14	Random withRepl(7)	35 samples	35	1,71%
15	Random withRepl(8)	40 samples	40	1,52%
16	Random withoutRepl(1)	5 samples	5	6,57%
17	Random withoutRepl(2)	10 samples	10	4,23%
18	Random withoutRepl(3)	15 samples	15	3,13%
19	Random withoutRepl(4)	20 samples	20	2,42%
20	Random withoutRepl(5)	25 samples	25	1,89%
21	Random withoutRepl(6)	30 samples	30	1,50%
22	Random withoutRepl(7)	35 samples	35	1,17%

	Method	Parameters	#Iter	AvgError
23	Random withoutRepl(8)	40 samples	40	0,90%
24	N-StepSampling(1)	N=10	7	5,81%
25	N-StepSampling(2)	N=8	8	5,51%
26	N-StepSampling(3)	N=7	10	3,42%
27	N-StepSampling(4)	N=6	11	3,34%
28	N-StepSampling(5)	N=5	13	3,99%
29	N-StepSampling(6)	N=4	16	3,31%
30	N-StepSampling(7)	N=3	22	2,07%
31	N-StepSampling(8)	N=2	32	1,55%
32	BinarySS-fix, withoutRepl(1)	1 sample/interval	10	1,76%
33	BinarySS-fix, withoutRepl(2)	2 samples/interval	16	0,96%
34	BinarySS-fix, withoutRepl(3)	3 samples/interval	21	0,75%
35	BinarySS-fix, withoutRepl(4)	4 samples/interval	25	0,40%
36	BinarySS-fix, withoutRepl(5)	5 samples/interval	29	0,31%
37	BinarySS-fix, withoutRepl(6)	6 samples/interval	32	0,25%
38	BinarySS-fix, withoutRepl(7)	7 samples/interval	34	0,20%
39	BinarySS-fix, withoutRepl(8)	8 samples/interval	36	0,14%
40	BinarySS-dyn, withoutRepl(1)	0,01*interval length	10	1,84%
41	BinarySS-dyn, without Repl(2)	0,02*interval length	12	1,71%
42	BinarySS-dyn, without Repl(3)	0,05*interval length	17	1,06%
43	BinarySS-dyn, without Repl(4)	0,07*interval length	23	0,67%
44	BinarySS-dyn, without Repl(5)	0,08*interval length	25	0,40%
45	BinarySS-dyn, without Repl(6)	0,11*interval length	28	0,32%
46	BinarySS-dyn, without Repl(7)	0,12*interval length	30	0,32%
47	BinarySS-dyn, without Repl(8)	0,13*interval length	34	0,18%
48	CurveFitting f1 (1)	15 random samples	16	3,11%

	Method	Parameters	#Iter	AvgError
49	CurveFitting f1 (2)	20 random samples	21	2,49%
50	CurveFitting f1 (3)	25 random samples	26	1,90%
51	CurveFitting f1 (4)	30 random samples	31	1,44%
52	CurveFitting f2 (1)	15 random samples	16	2,37%
53	CurveFitting f2 (2)	20 random samples	21	1,73%
54	CurveFitting f2 (3)	25 random samples	26	1,41%
55	CurveFitting f2 (4)	30 random samples	31	1,13%
56	CurveFitting f3 (1)	20 random samples	21	2,18%
57	CurveFitting f3 (2)	25 random samples	26	1,52%
58	CurveFitting f3 (3)	30 random samples	31	1,36%
59	CurveFitting f4 (1)	20 random samples	20	2,14%
60	CurveFitting f4 (2)	25 random samples	25	1,64%
61	CurveFitting f4 (3)	30 random samples	30	1,37%
62	BinarySS(fix, without Repl +Curve fitting f2 (1)	f2, 1 sample/interval	11	1,29%
63	BinarySS(fix, without Repl +Curve fitting f2 (2)	f2, 2 samples/interval	17	0,69%
64	BinarySS(fix, without Repl +Curve fitting f2 (3)	f2, 3 samples/interval	22	0,51%
65	BinarySS(fix, without Repl +Curvefitting f2 (4)	f2, 4 samples/interval	26	0,35%
66	BinarySS(fix, without Repl +Curve fitting f2 (5)	f2, 5 samples/interval	29	0,29%
67	BinarySS(fix, without Repl +Curvefitting f2 (6)	f2, 6 samples/interval	33	0,22%

Figure 4: Aggregated Results

cation on every platform. Using over one hundred different workloads helps test the robustness and prediction accuracy of the described algorithms, and reduces the probability of (but does not exclude) outlier workloads in which all of the described algorithms would behave in an entirely different way.

6. AUTOTUNIUM – AN EXTENSIBLE PERFORMANCE TUNER

How can software engineers take advantage of the insights in this paper? To allow auto-tuners be used easily in everyday practice, we developed Autotunium, an auto-tuner that can be extended with user-defined tuning algorithms. It is implemented in Java an runs as an Eclipse plugin to help programmers implement and tune parallel applications directly in a development environment. Alternatively, Autotunium can be run as a stand-alone application or from the command line, which is useful when applications are re-tuned after deployment or migration to new multicore platforms. Autotunium has been tested on Windows, Linux, and MacOS.

A special feature of Autotunium its plugin extensibility, e.g., with tuning algorithms and so-called evaluators that provide run-time feedback to the auto-tuner. For example, an evaluator can measure the execution time or speedup of a whole program and pass the value to an algorithm plugin via a predefined interface. The tuning algorithm designer can fully concentrate on the essential algorithmic issues without implementing an entire infrastructure. It is thus possible to create libraries with tuning algorithms that can be shared and accessed by a broad range of developers.

Figure 6 shows the application configuration screen, which allows developers to specify the tuning parameters and their ranges. For now, Autotunium assumes that tunable programs have numeric command line parameters for every tunable parameter, and that the ranges of these parameters are defined either via the graphical user interface or via a configuration file. Autotunium can also generate specific project-related command lines with information gathered from the Eclipse IDE.

With Autotunium, we also applied multidimensional autotuning successfully using several parameters and parameter types other than threads count. For example, we applied Binary Search Sampling to PBzip2, and simultaneously tuned thread count as a parameter in one dimension, and compression block size as a second parameter in another dimension.

Figure 7 shows the algorithm configuration screen, where

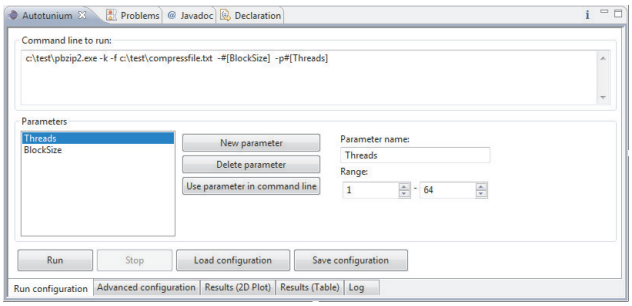

Figure 6: Autotunium application configuration.

the user specifies the optimization details, such as which algorithm to use, optimization goal, algorithm-specific parameters, evaluator to use, and maximum timeouts for program execution. In addition, an external directory for plugin extensions can be specified (e.g., for plugins that users might download from the Internet). Autotunium also supports a simulation mode for testing purposes, which uses workload data that was previously stored in a file instead of gathering run-time data from an executing program.

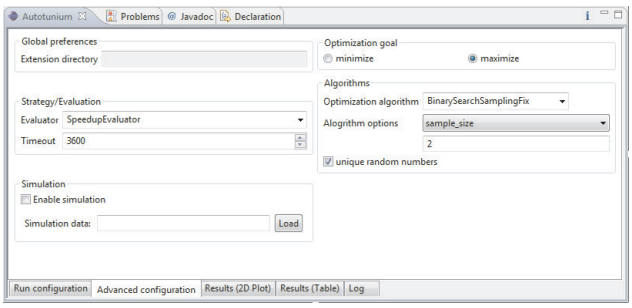

Figure 7: Autotunium algorithm configuration.

The results of each tuning iteration are logged and can be viewed as a graph or in tabular form. All values can be exported and analyzed with external programs. Figure 8 depicts tuning results in tabular form for measurements executed on a 4-core machine, along with a highlighted parameter combination that leads to the best speedup. The tuning results show on that particular machine that compressing a

20MB binary file with PBzip2 has the best speedup at 28 threads and a block size of 500KB.

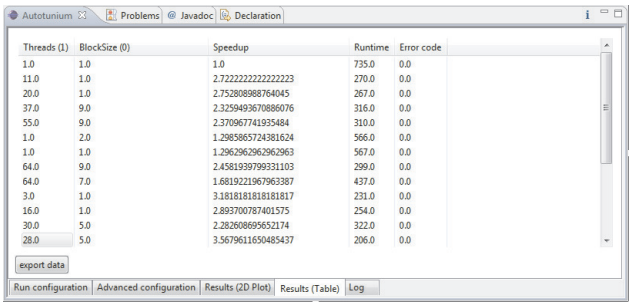

Figure 8: Autotunium tabular results.

Autotunium allows the developer to explore the multidimensional search space graphically. The first screenshot in Figure 9 shows the sampled thread count and block size value combinations. The second screenshot shows another dimensional cut, which is speedup against number of threads. The third screenshot illustrates yet another cut, which is the speedup against the block size (*100 KB) used for compression.

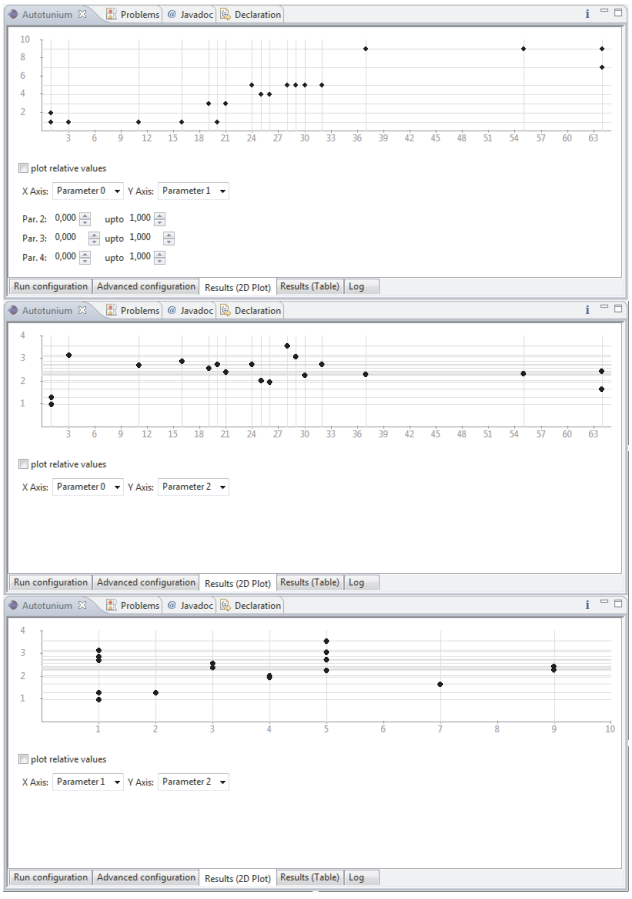

Figure 9: Autotunium graphical analysis.

Programmers can directly interact with Autotunium's graphical outputs to better understand the application behavior and its performance on a certain platform. This is meant to speed up developer productivity.

7. RELATED WORK

Most of the related work is domain-specific and has been done in numerics and large-scale parallel applications on clusters. Key approaches, libraries, and tools are discussed in [3]. In contrast to our approach, auto-tuners like ATLAS [26], Spiral [20], FFTW [6] focusing on numerical programs also generate different code versions (e.g., for Fourier Transform) that are tried out by an auto-tuner (e.g., with different loop unrollings or buffer sizes). In our approach, we aim to make auto-tuning easier applicable to all sorts of multicore applications without imposing a particular type of application. An environment for dynamic tuning is discussed in [15], however, without evaluating tuning algorithms. A run-time tuning approach targeted at Grid environments is presented in [5]. In [10], the FIBER tuning facility is presented for clusters and adapted to an eigensolver, with tuning based on random sampling and fitting 5th order polynomials. The need for auto-tuners on multicore platforms is advocated in [2]. A pattern-based search is introduced in [21], but the improvements are only 3% better than random sampling; by contrast, our approach of binary search sampling combined with curve fitting reduces the error by an order of magnitude, compared to random sampling. Compiler flag combinations that lead to best speedups on dual-core machines are evaluated in [4], and [7] presents a collective tuning infrastructure in which users can exchange data relevant for performance optimization. Compiler optimizations for automatic performance tuning on SPARC II and Pentium IV platforms are discussed in [16]. An approach for procedure-level autotuning for compiler optimizations is introduced in [17]. Direct search methods are theoretically contrasted in [12]. A method to automatically tune the number of processors and optimize loops has been proposed in [25] for applications using the SPMD programming model. Machine learning tuning techniques are used in [22] on clusters to predict numerical program performance based on application-specific models.

8. CONCLUSION

This paper is among the first to quantify the effectiveness of auto-tuning algorithms on a large set of multicore applications running on contemporary desktops and servers. The results show that applications need to be re-tuned on new multicore platforms. Classical optimization approaches such as hill climbing and simulated annealing don't work well for finding the optimum number of application threads. In contrast, the proposed combination of binary search sampling and non-linear modeling reduces errors by an order of magnitude. The approach is effective and at the same time does not require input-, application- or platform-specific models. This way, it avoids altogether the state-space explosion problem that appears when many platform configurations exist.

Acknowledgements

We would like to thank Andreas Zwinkau, Martin Heneka, Thomas Karcher for their implementation work Autotunium, and the Excellence Initiative and the Karlsruhe Institute of Technology for financial support.

9. REFERENCES

[1] S. G. Akl. *The Design and Analysis of Parallel Algorithms*. Prentice Hall, 1989.

[2] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 18 2006.

[3] D. H. Bailey, R. Lucas, P. Hovland, B. Norris, K. Yelick, D. Gunter, B. de Supinski, D. Quinlan, P. Worley, J. Vetter, P. Roth, O. R. N. Laboratory, J. Mellor-Crummey, A. Snavely, J. Hollingsworth, D. Reed, R. Fowler, Y. Zhang, M. Hall, J. Chame, J. Dongarra, and S. Moore. Performance engineering: Understanding and improving the performance of large-scale codes. *CTWatch Quarterly*, 3(4), 2007.

[4] Y. Chen, Y. Huang, L. Eeckhout, G. Fursin, L. Peng, O. Temam, and C. Wu. Evaluating iterative optimization across 1000 datasets. In *PLDI '10: Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, pages 448–459, New York, NY, USA, 2010. ACM.

[5] C. Ţăpuş, I.-H. Chung, and J. K. Hollingsworth. Active harmony: towards automated performance tuning. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–11, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

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

[7] G. Fursin. Collective tuning initiative: automating and accelerating development and optimization of computing systems. In *Proceedings of the GCC Summit'09*, Montreal, Canada, 2009.

[8] J. Gilchrist. Parallel bzip2. http://compression.ca/pbzip2, 2010.

[9] H. H. Hoos and T. Stuetzle. *Stochastic Local Search*. Morgan Kaufmann, 2005.

[10] T. Katagiri, K. Kise, H. Honda, and T. Yuba. Fiber: A generalized framework for auto-tuning software. In *International Symposium High Performance Computing (ISHPC)*, number 2858 in LNCS, pages 146–159, 2003.

[11] S. Kirkpatrick, C. D. G. Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 13 1983.

[12] R. M. Lewis, V. Torczon, and M. W. Trosset. Direct search methods: then and now. *J. Comput. Appl. Math.*, 124(1-2):191–207, 2000.

[13] D. Meder. Examples for multithreaded program implementations. Course Material. Karlsruhe Institute of Technology, 2009.

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

[15] A. Morajko, T. Margalef, and E. Luque. Design and implementation of a dynamic tuning environment. *J. of Par. and Distr. Comp.*, 67(4), 2007.

[16] Z. Pan and R. Eigenmann. Rating compiler optimizations for automatic performance tuning. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 14, Washington, DC, USA, 2004. IEEE Computer Society.

[17] Z. Pan and R. Eigenmann. Fast, automatic, procedure-level performance tuning. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 173–181, New York, NY, USA, 2006. ACM.

[18] V. Pankratius, A. Jannesari, and W. Tichy. Parallelizing bzip2: A case study in multicore software engineering. *Software, IEEE*, 26(6), nov.-dec. 2009.

[19] V. Pankratius, C. Schaefer, A. Jannesari, and W. F. Tichy. Software engineering for multicore systems: an experience report. In *Proc. ACM IWMSE '08*, 2008.

[20] M. Püschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. Spiral: code generation for dsp transforms. *Proceedings of the IEEE*, 93(2), 2005.

[21] A. Qasem, K. Kennedy, and J. Mellor-Crummey. Automatic tuning of whole applications using direct search and a performance-based transformation system. *J. Supercomput.*, 36(2), 2006.

[22] K. Singh, E. Ipek, S. A. McKee, B. R. de Supinski, M. Schulz, and R. Caruana. Predicting parallel application performance via machine learning approaches. *Conc. and Comp.: Pract. and Exp.*, 19(17), 2007.

[23] J. Stone. Tachyon parallel / multiprocessor ray tracing system. http://jedi.ks.uiuc.edu/ johns/raytracer/, January 2010.

[24] S. K. Thomson. *Sampling*. John Wiley and Sons, 1992.

[25] O. Werner-Kytölä and W. F. Tichy. Self-tuning parallelism. In *HPCN Europe 2000: Proceedings of the 8th International Conference on High-Performance Computing and Networking*, pages 300–312, London, UK, 2000. Springer-Verlag.

[26] C. R. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1-2):3–35, January 2001.

[27] Wolfram Research. Mathematica v. 7. www.wolfram.com, 2010.

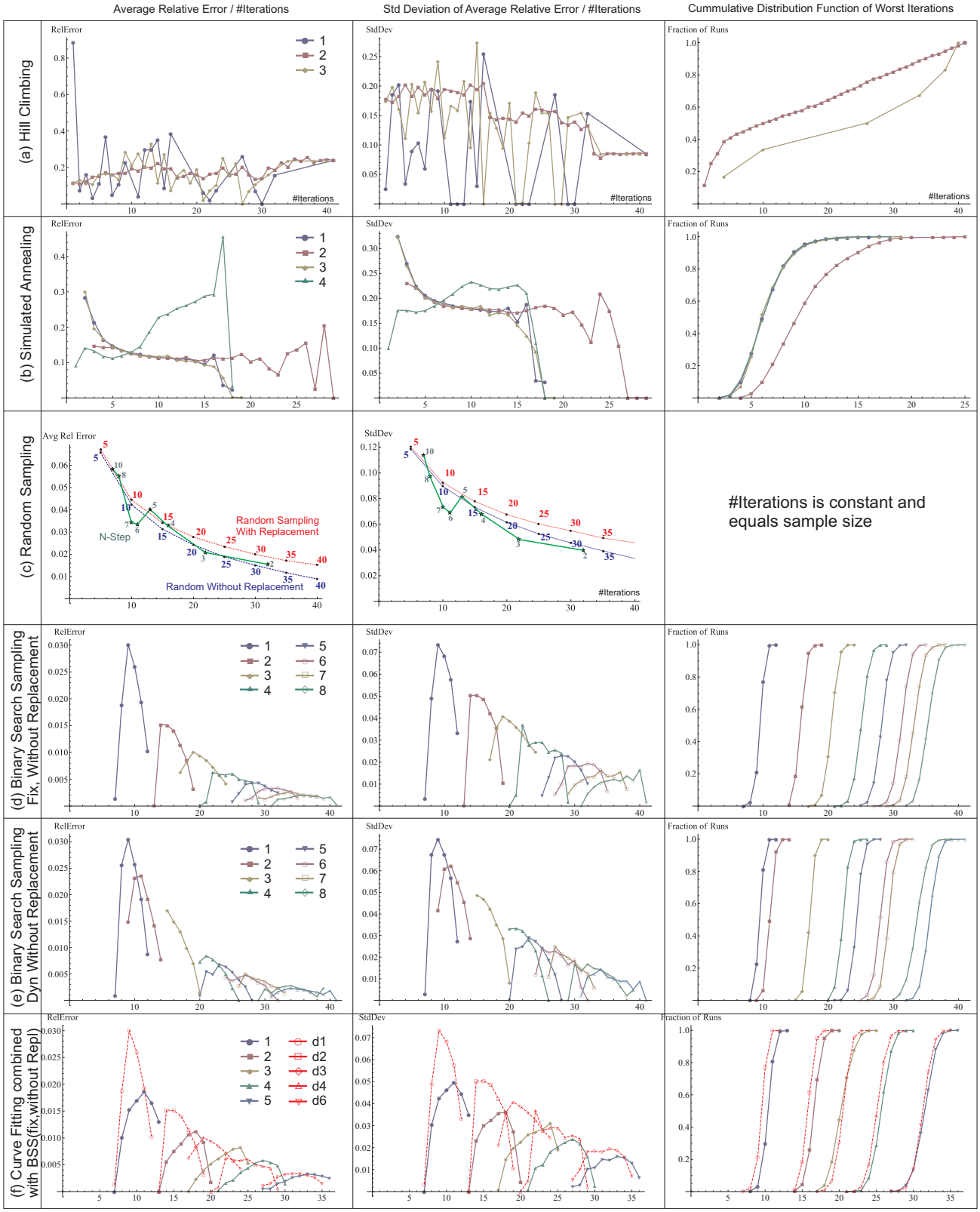

Figure 5: Empirical distributions for errors, standard deviations, and number of iterations. The numbers within the graphs refer to Figure 4.