Evolutionary Algorithms for Compiler-Enabled Program Autotuning

by

Maciej Pacula

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2011

© Massachusetts Institute of Technology 2011. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
May 19, 2011

Certified by . . . .
Una-May O'Reilly
Principal Research Scientist
Thesis Supervisor

Accepted by . . . .
Dr. Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

Evolutionary Algorithms for Compiler-Enabled Program Autotuning

by

Maciej Pacula

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 2011, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

PetaBricks [4, 21, 7, 3, 5] is an implicitly parallel programming language which, through the process of *autotuning*, can automatically optimize programs for fast QoS-aware execution on any hardware. In this thesis we develop and evaluate two PetaBricks autotuners: INCREA and SiblingRivalry. INCREA, based on a novel bottom-up evolutionary algorithm, optimizes programs offline at compile time. SiblingRivalry improves on INCREA by optimizing online during a program's execution, dynamically adapting to changes in hardware and the operating system. Continuous adaptation is achieved through *racing*, where half of available resources are devoted to always-on learning. We evaluate INCREA and SiblingRivalry on a large number of real-world benchmarks, and show that our autotuners can significantly speed up PetaBricks programs with respect to many non-tuned and mis-tuned baselines. Our results indicate the need for a continuous learning loop that can optimize efficiently by exploiting online knowledge of a program's performance. The results leave open the question of how to solve the online optimization problem on all cores, i.e. without racing.

Thesis Supervisor: Una-May O'Reilly
Title: Principal Research Scientist

Acknowledgments

The work presented in this thesis would not have been possible without the help of many people. I am especially grateful to my advisor, Una-May O'Reilly, who introduced me to the PetaBricks project before I even knew what autotuning was, and whose endless enthusiasm and extensive knowledge of Evolutionary Computation inspired and guided me through my Senior and M.Eng. years at MIT. Many of the ideas presented in this thesis were a result of our fruitful and engaging discussions.

I would also like to thank Jason Ansel, the original author of PetaBricks, for his insightful ideas and substantial contributions to many of the sections in this thesis. I am also grateful to Saman Amarasinghe for his advice and impressive knowledge of compilers and computer architecture, Marek Olszewski for his help writing some of the SiblingRivalry sections and running power consumption experiments, and many other members of the Commit group for their helpful suggestions and work on the PetaBricks language. In addition, portions of this thesis were originally described and reported in papers with multiple co-authors, namely [5] and [6].

Finally, I would like to thank my girlfriend Liz, as well as friends and family for their love, patience and support. This work would not have been possible without you.

The work presented in this thesis is supported by DOE Award DE-SC0005288.

Contents

1 Introduction **19**

1.1 Contributions 21

1.2 Thesis Outline 22

2 Background **23**

2.1 Autotuning 23

2.2 Evolutionary Algorithms 25

2.2.1 Problem-Specific Components 28

2.2.2 Representation-Dependent Components 30

2.2.3 General Components 31

2.2.4 Genetic Algorithms 33

2.2.5 Multi-objective Algorithms 35

2.3 The PetaBricks Language 35

2.3.1 Example PetaBricks Program: kmeans 36

2.3.2 Variable Accuracy Algorithms 37

2.4 PetaBricks Autotuning 41

2.4.1 Properties of the Autotuning Problem 43

2.5 Benchmarks 44

2.5.1 Fixed Accuracy 44

2.5.2 Variable Accuracy 45

3 Offline Autotuning **47**

3.1 General-Purpose EA (GPEA) 48

3.1.1 Representation 48
3.1.2 Initialization 49
3.1.3 Fitness Evaluation 49
3.1.4 Variation Operators 50
3.1.5 Parent and Survivor Selection 51
3.1.6 Termination Condition 51
3.2 Bottom-Up EA (INCREA) 51
3.3 Representation 51
3.3.1 Top level Strategy 52
3.3.2 Mutation Operators 53
3.3.3 Dealing with Noisy Fitness 55
3.4 Experimental Results 57
3.4.1 Experimental Setup 57
3.4.2 INCREA vs GPEA 57
3.4.3 Representative Runs 60
3.5 Conclusions 65

4 Online Autotuning 67

4.1 Competition Execution Model 69
4.1.1 Other Splitting Strategies 70
4.1.2 Time Multiplexing Races 70
4.2 SiblingRivalry 72
4.2.1 High Level Function 72
4.2.2 Selecting the Safe and Seed Configuration 75
4.2.3 Mutation Operators 76
4.2.4 Adaptive Mutator Selection (AMS) 77
4.2.5 Credit Assignment 78
4.2.6 Bandit Mutator Selection 78
4.2.7 Population Pruning 79
4.3 Related Work 80

4.4 Experimental Results 82
4.4.1 Experimental Setup 82
4.4.2 Sources of Speedups 82
4.4.3 Load on a System 83
4.4.4 Migrating Between Microarchitectures 87
4.4.5 Cold Start 91
4.4.6 Power Consumption 93
4.4.7 Conclusions 93

5 Hyperparameter Tuning **95**
5.1 Tuning the Tuner 95
5.2 Hyperparameter Quality 97
5.2.1 Static System 98
5.2.2 Dynamic System 98
5.3 Experimental Results 99
5.3.1 Sort 101
5.3.2 Bin Packing 101
5.3.3 Poisson 102
5.3.4 Image Compression 104
5.4 The Big Picture 104
5.4.1 Globally Optimal Hyperparameters 105
5.5 Conclusions 105

6 Conclusions and Future Work **115**

A Detailed Statistics **123**
A.1 Hyperparameters Runs: Normality Testing 123
A.1.1 Xeon8 123
A.1.2 AMD48 133
A.2 Hyperparameter Runs: Significance Testing 143
A.2.1 Xeon8 144

A.2.2 AMD48 153

List of Figures

1-1 The STL `std::sort` routine. Insertion sort is used for inputs smaller than 15 elements, and merge sort is used for larger inputs. The 15-element cutoff is hard-coded into the library. From G++ 4.4 headers included with Ubuntu 10.10. 20

2-1 A functional diagram of an Evolutionary Algorithm. The algorithm evaluates candidate solutions using a problem-specific fitness function f, and produces new solutions using selection and variation operators, which differ by algorithm (adapted from `http://groups.csail.mit.edu/EVO-DesignOpt/uploads/Site/evoopt.png`). 27

2-2 An example run of the `kmeans` algorithm on a set of 2-D points (`Points`), with the number of clusters fixed at 3. The crosshairs mark cluster centers (`Centroids`) and different point colors (`Assignments`) correspond to different clusters. 38

2-3 PetaBricks pseudocode for `kmeans` 39

2-4 Dependency graph for `kmeans` example. The rules are the vertices while each edge represents the dependencies of each rule. Each edge color corresponds to each named data dependence in the pseudocode. 40

2-5 A selector for a sample sorting algorithm where $\mathbf{C_s} = [150, 10^6]$ and $\mathbf{A_s} = [1, 2, 0]$. The selector selects the *InsertionSort* algorithm for input sizes in the range $[0; 150)$, *QuickSort* for input sizes in the range $[150, 10^6)$ and *RadixSort* for $[10^6, MAXINT)$. *BitonicSort* was suboptimal for all input ranges and is not used. 42

3-1 A sample genome for $m = 2$, $k = 2$ and $n = 4$. Each gene stores either a cutoff $c_{s,i}$, an algorithm $\alpha_{s,i}$ or a tunable value t_i. 52

3-2 Top level strategy of INCREA. 54

3-3 Pseudocode of function "fitter". 56

3-4 Execution time for target input size with best individual of generation. Mean and standard deviation (shown in error bars) with 30 runs. . . 59

3-5 Time out and population growth statistics of INCREA for 30 runs of Sort on target input size 2^{20}. Error bars are mean plus and minus one standard deviation. 60

3-6 Representative runs of INCREA and GPEA on each benchmark. The left graphs plot the execution time (on the target input size) of the best solution after each generation. The right graph plots the number of fitness evaluations conducted at the end of each generation. All graphs use seconds of training time as the x-axis. 61

4-1 High level flow of the runtime system. The data on dotted lines may not be transmitted for the slower configuration, which can be terminated before completion. 69

4-2 Pseudocode of how requests are processed by the online learning system 74

4-3 The credit assigned to mutator μ is the area under the curve. Section 4.2.5 provides details. Reproduced from [29]. 79

4-4 Speedups (or slowdowns) of each benchmark as the load on a system changes. Note that the 50% load and 100% load speedups for Clustering in (b), which were cut off due to the scale, are 4.0x and 3.9x. . . . 84

4-5 Representative graphs for varying system load showing throughput over time. Benchmark is LU Factorization on AMD48. 85

4-6 The scenario with frequent migration modeled by our architecture migration experiments. We compare a fixed configuration (found with offline training on a different machine) to SiblingRivalry, to show how adapting to each architecture can improve throughput. We measure throughput only between the first and second migration, and include the cost of all learning in the throughput measurements. 87

4-7 Speedups (or slowdowns) of each benchmark after a migration between microarchitectures. "Normalized throughput" is the throughput over the first 10 minutes of execution of SiblingRivalry (including time to learn), divided by the throughput of the first 10 minutes of an offline tuned configuration using the entire system. 88

4-8 Representative graphs of throughput over time for fixed accuracy benchmarks after a migration between microarchitectures. 89

4-9 Representative graphs of throughput over time for variable-accuracy benchmarks after a migration between microarchitectures. "Target" is the accuracy target both the offline and online tuners are set to optimize for. 90

4-10 The effect of using an offline tuned configuration as a starting point for SiblingRivalry on the Sort benchmark. We compare starting from a random configuration ("w/o offline") to configurations found through offline training on the same and a different architecture. 91

4-11 Average energy use per request for each benchmark after migrate Xeon8 to AMD48. 92

5-1 Best performing hyperparameters and associated score function values under the Static System and Dynamic System autotuning scenarios on `Xeon8` and `AMD48` architectures. 100

5-2 Metrics for benchmark Sort on the `Xeon8` system evaluated with different values of hyperparameters. An asterisk $*$ next to a number means that the difference from optimum is not statistically significant (p-value ≥ 0.05). 107

5-3 Metrics for benchmark Sort on the `AMD48` system evaluated with different values of hyperparameters. An asterisk $*$ next to a number means that the difference from optimum is not statistically significant (p-value ≥ 0.05). 108

5-4 Metrics for benchmark Bin Packing on the `Xeon8` system evaluated with different values of hyperparameters. An asterisk $*$ next to a number means that the difference from optimum is not statistically significant (p-value ≥ 0.05). 109

5-5 Metrics for benchmark Bin Packing on the `AMD48` system evaluated with different values of hyperparameters. An asterisk $*$ next to a number means that the difference from optimum is not statistically significant (p-value ≥ 0.05). 110

5-6 Metrics for benchmark Poisson on the `Xeon8` system evaluated with different values of hyperparameters. An asterisk $*$ next to a number means that the difference from optimum is not statistically significant (p-value ≥ 0.05). 111

5-7 Metrics for benchmark Poisson on the `AMD48` system evaluated with different values of hyperparameters. An asterisk $*$ next to a number means that the difference from optimum is not statistically significant (p-value ≥ 0.05). 112

5-8 Metrics for benchmark Image Compression on the `Xeon8` system evaluated with different values of hyperparameters. An asterisk $*$ next to a number means that the difference from optimum is not statistically significant (p-value ≥ 0.05). 113

5-9 Metrics for benchmark Image Compression on the AMD48 system evaluated with different values of hyperparameters. An asterisk $*$ next to a number means that the difference from optimum is not statistically significant (p-value ≥ 0.05). 114

List of Tables

2.1 INCREA and SiblingRivalry compared to state-of-the-art autotuners from literature. 25

2.2 Listing of benchmarks and their properties. 44

3.1 INCREA and GPEA Parameter Settings. 57

3.2 Comparison of INCREA and GPEA in terms of mean time to convergence in seconds and in terms of execution time of the final configuration. Standard deviation is shown after the ± symbol. The final column is statistical significance determined by a t-test (lower is better). 58

3.3 Listing of the best genome of each generation for each autotuner for an example training run. The genomes are encoded as a list of algorithms (represented by letters), separated by the input sizes at which the resulting program will switch between them. The possible algorithms are: I = insertion-sort, Q = quick-sort, R = radix-sort, and M^x = x-way merge-sort. Algorithms may have a $_p$ subscript, which means they are run in parallel with a work stealing scheduler. For clarity, unreachable algorithms present in the genome are not shown. 63

3.4 Effective and ineffective mutations when INCREA solves Sort (target input size 2^{20}.) 63

4.1 Specifications of the test systems used and the acronyms used to differentiate them in results. 82

5.1 Benchmark scores for the globally optimal values of hyperparameters normalized with respect to the best score for the given benchmark and scenario. The hyperparameters were $C = 5$, $W = 5$ for the Static System, and $C = 5$, $W = 100$ for the Dynamic System. Mean scores are 0.8832 and 0.8245 for the Static and Dynamic systems, respectively. 105

Chapter 1

Introduction

Despite the ever-increasing processing power of modern computers, high performance computation remains a commodity. A prime example are cloud services offered by companies such as Amazon, Google and Microsoft, where the user is billed depending on the desired CPU resources. The use of such resources also affects energy consumption, which is important in systems ranging from embedded devices to large data centers [13]. For these reasons, users of computer software often demand efficient resource utilization, and a program that can perform the same work in less time, while also meeting some Quality of Service (QoS) guarantee, is usually considered better.

Unfortunately, optimizing software for optimal performance is a difficult feat and carries with it multiple caveats [3, 4]. While modern compilers attempt to take some of the optimization burden off the programmer, they are usually only successful at optimizing single algorithms and even then the range of possible optimizations is limited [4]. In many applications, such as sorting, matrix multiplication and multigrid solvers significant performance boosts can be achieved by constructing hybrid algorithms, where the appropriate algorithm is chosen depending on the size of the input and hardware characteristics. However, the burden is on the programmer to incorporate such hybrid algorithms, manually writing glue code and determining under what conditions the given algorithm should be invoked. Today's compilers are unable to automate this process because of their reliance on traditional, low-level control structures such as loops and switches [3, 4].

What's worse, it is often impossible to obtain a universal, "one-size-fits-all" solution that achieves optimal performance on all hardware configurations in all contexts. As such, software with hard-coded and often suboptimal algorithmic compositions is commonplace. An example can be found in the popular C++ Standard Template Library (STL) (Figure 1-1), whose `sort` routine uses insertion sort for inputs smaller than 15 elements and merge sort for inputs larger than 15 elements. Tests show that much larger cutoffs perform better on modern architectures [4].

```
1   template<typename _RandomAccessIterator>
2   void
3   __inplace_stable_sort(_RandomAccessIterator __first,
4   _RandomAccessIterator __last)
5   {
6     if (__last - __first < 15)
7     {
8       std::__insertion_sort(__first, __last);
9       return;
10    }
11    _RandomAccessIterator __middle = __first + (__last - __first)
          / 2;
12    std::__inplace_stable_sort(__first, __middle);
13    std::__inplace_stable_sort(__middle, __last);
14    std::__merge_without_buffer(__first, __middle, __last,
15      __middle - __first,
16      __last - __middle);
17  }
```

Figure 1-1: The STL `std::sort` routine. Insertion sort is used for inputs smaller than 15 elements, and merge sort is used for larger inputs. The 15-element cutoff is hard-coded into the library. From G++ 4.4 headers included with Ubuntu 10.10.

Unsurprisingly, automatic optimization of computer programs has been an active area of research. PetaBricks [4, 21, 7, 3, 5] is a is an implicitly parallel programming language for high performance computing which aims to solve the problems described above. It provides language constructs to naturally express algorithmic compositions through the concept of *algorithmic choices*. The programmer can simply state what algorithms are applicable at the given point of the program, letting the compiler generate the necessary glue and decide how the algorithms should be composed. In addition to algorithmic choices, PetaBricks also allows the programmer to

define tunable parameters such as blocking sizes and the number of worker threads, whose optimal value is up to the compiler to select. The process of automatically determining algorithmic compositions and the values of tunables is called *autotuning*, and the autotuning program is called an *autotuner*. The autotuner's goal is to find program configurations which maximize speed while meeting Quality of Service (QoS) guarantees.

1.1 Contributions

In this thesis we develop, evaluate and compare two PetaBricks autotuners: INCREA and SiblingRivalry. We provide experimental results for a large number of real-world benchmarks on a number of different architectures under different conditions.

INCREA makes the following contributions:

- It introduces a novel evolutionary algorithm for solving problems where evaluation is expensive and noisy.
- It can take advantage of shortcuts based on problem properties by reusing solutions to smaller problem instances when solving larger problems.
- It demonstrates that incremental solving works well on real-world problems.

In addition, SiblingRivalry makes the following contributions:

- To the best of our knowledge, the first general technique to apply evolutionary tuning algorithms to the problem of online autotuning of computer programs.
- A new model for online autotuning where the processor resources are divided and two candidate configurations compete against each other.
- A multi-objective, practical online evolutionary learning algorithm for high-dimensional, multi-modal, and non-linear configuration search spaces.

- A scalable learning algorithm for high-dimensional search spaces, such as those in our benchmark suite which average 97 search dimensions.
- Support for meeting dynamically changing time or accuracy targets which are in response to changing load or user requirements.
- Experimental results showing a geometric mean speedup of 1.8x when adapting to changes in microarchitectures and a 1.3x geometric mean speedup when adapting to moderate load on the system.
- Experimental results showing how, despite accomplishing more work, SiblingRivalry can actually reduce average power consumption by an average of 30% after a migration between microarchitectures.

1.2 Thesis Outline

The remainder of the thesis is organized as follows. Chapter 2 provides background on the PetaBricks language, the autotuning problem, evolutionary algorithms and the benchmarks used to evaluate autotuners. Chapter 3 presents and experimentally evaluates the offline autotuner INCREA. Chapter 4 describes the SiblingRivalry online autotuner and evaluates its performance on multiple architectures under different conditions. Chapter 5 performs an in-depth evaluation of SiblingRivalry's sensitivity to hyperparameters. Finally, Chapter 6 draws conclusions.

Chapter 2

Background

2.1 Autotuning

For the purposes of this thesis, *autotuning* is a process of optimizing algorithmic choices and tunable parameters of a program in order to achieve the fastest possible execution while meeting desired Quality of Service guarantees. We can classify different approaches to autotuning with respect to a number of independent criteria:

- generality: algorithm-specific vs. general-purpose
- hardware optimization capability: hardware-aware vs. hardware-oblivious
- number of objectives: single vs. multi-objective
- model dependence: model-based vs. model-free
- tuning process: online vs. offline

Generality of an autotuner specifies whether it can autotune arbitrary programs, or only a limited set of algorithms. For example, an algorithm-specific autotuner might be designed to optimize only the matrix multiplication algorithm in a specific math library. General-purpose autotuners can optimize any program written in a given language, and the programs that will be autotuned are generally not known when the autotuner is being implemented.

Hardware optimization capability of an autotuner defines whether it can adapt to different hardware configurations. Hardware-oblivious autotuners perform optimizations that have a chance of improving performance independent of the machine that the tuned program runs on, but cannot take advantage of hardware-specific features such as hyperthreading, high memory bandwidth, or many others. Hardware-aware autotuners, on the other hand, can exploit both hardware-independent as well as hardware-specific optimizations.

Single-objective autotuners optimize only the running time of the program, attempting to produce the fastest possible executable, but cannot take advantage of possible beneficial trade-offs with other objectives inherent in the problem. For example, a single-objective autotuner cannot deliberately sacrifice accuracy to gain speed and vice-versa. Multi-objective autotuners, on the other hand, are built with such trade-offs in mind and can provide significant speedups by, for example, detecting when a program exceeds a user-defined accuracy target and decreasing the accuracy accordingly, saving execution time. While in principle multi-objective autotuners can deal with any number of objectives, in this thesis we limit ourselves to time and accuracy.

Model-based autotuners create a model of the program (and sometimes hardware) they tune and use it to their advantage. For example, a model-based autotuner might use the model to estimate the running time of a given program without actually running it, saving computation. However, such models are at best only an approximation of reality and are designed with a number of assumptions in mind. Whenever those assumptions do not hold, the autotuner runs the risk of not being able to find the optimal configuration. Model-free autotuners, on the other hand, do not build models and instead optimize the tuned program directly. As a result, any execution intricacies, including those not predicted by the autotuner's designers, can potentially be exploited.

Offline autotuners optimize their program once, usually at compile time, and re-use that static configuration throughout the lifetime of the program. Offline autotuning can be burdensome to the deployment of a program, since the tuning process

Autotuner	General-Purpose	Hardware-Aware	Multi-Objective	Model-Free	Online
INCREA	**Yes**	**Yes**	No	**Yes**	No
SiblingRivalry	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
ATLAS[53]	No	**Yes**	No	**Yes**	No
FFTW[31]	No	**Yes**	No	No	No
Green[13]	**Yes**	**Yes**	**Yes**	No	**Yes**
PowerDial[35]	**Yes**	**Yes**	**Yes**	No	**Yes**

Table 2.1: INCREA and SiblingRivalry compared to state-of-the-art autotuners from literature.

can take a long time and should be re-run whenever the program, microarchitecture, execution environment, or tool chain changes. Failure to re-autotune programs often leads to widespread use of sub-optimal algorithms. With the growth of cloud computing, where computations can run in environments with unknown load and migrate between different (possibly unknown) microarchitectures, the problems with offline autotuners become even more apparent. In contrast, online autotuners are always-on and can dynamically adapt to changes, automatically re-tuning a program when necessary.

Table 2.1 compares, along the above criteria, INCREA and SiblingRivalry against popular state-of-the-art autotuners described in literature.

2.2 Evolutionary Algorithms

The two autotuners presented in this thesis are based on Evolutionary Algorithms (EAs). Evolutionary Algorithms are stochastic global optimization methods that mimic Darwinian evolution, utilizing concepts such as inheritance, crossover, mutation and "survival of the fittest" [29]. EAs are a subset of a broader class of nature-inspired optimization algorithms which also include Artificial Neural Networks (ANNs), Particle Swarm Optimization (PSO), Ant Colony Optimization (ACO), and Simulated Annealing (SA), among others.

Within the domain of Evolutionary Algorithms, seminal work focused on techniques known as Genetic Algorithms (GAs), Evolution Strategies (ES), Evolutionary Programming (EP) and Genetic Programming (GP). Some other popular techniques

were introduced more recently, and include Differential Evolution (DE), and multi-objective evolutionary algorithms such as NSGA-II [26, 29, 28, 37]. Out of the above, GAs are described in more detail in Section 2.2.4.

Evolutionary Algorithms have been shown to be an effective optimization method for many problems where standard approaches failed. They can efficiently deal with vast search landscapes, landscapes in which the objective function is not differentiable and/or not well-specified (black box approaches), and can be robust in the face of a noisy objective function. The success of EAs in many difficult problem domains can be attributed to the large number of available techniques, and adjustable parameters in each that can be tailored to particular use cases [29].

Despite their differences, different EA methods follow a similar high-level approach (Figure 2-1), which can be summarized as follows:

1. A set of candidate solutions (the *population*) is generated, or reused from the previous *generation* (previous run of this loop).

2. The quality of candidate solutions is evaluated by a *fitness function*, which provides a numeric quality measure for each candidate (or multiple measures in the multi-objective case).

3. The fitness information from the previous step is used to select *parent* solutions in a process termed *selection*. In general, the highest the fitness value of a candidate, the higher its chance of being selected.

4. Parent solutions are then subjected to *variation operators* such as crossover and mutation, which modify them slightly and thus generate new candidate solutions, the *offspring*. These offspring, and sometimes certain parents, become the new population.

5. The process is repeated until some stop condition is reached, e.g. a candidate solution has been found which maximizes the fitness function or a specified number of generations has elapsed.

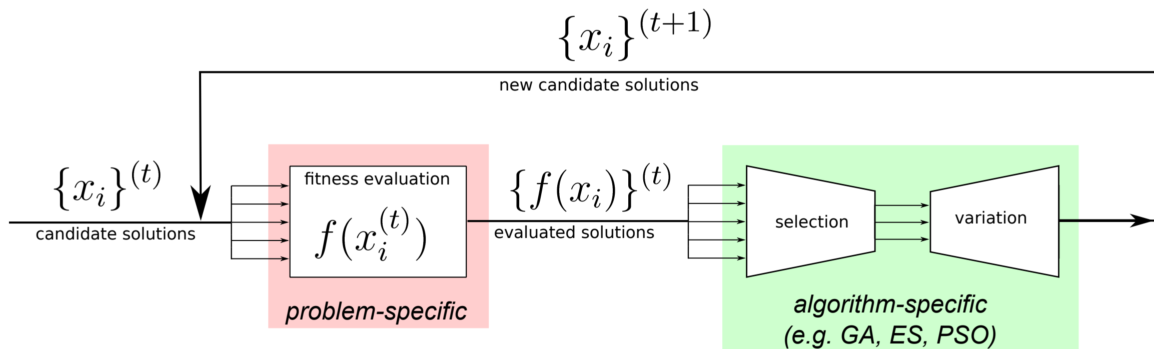


Figure 2-1: A functional diagram of an Evolutionary Algorithm. The algorithm evaluates candidate solutions using a problem-specific fitness function f, and produces new solutions using selection and variation operators, which differ by algorithm (adapted from `http://groups.csail.mit.edu/EVO-DesignOpt/uploads/Site/evoopt.png`).

The variation operators are responsible for exploring the search space by introducing random variation into the current population [29]. By applying these operators only to the fittest individuals, we hope to explore only solutions which have the potential to be better than their parents. This is based on the *locality* assumption: small variation to a given candidate should produce small variations in its fitness. The selection component ensures that we do not accept solutions which are much worse than other candidate solutions in the population.

It is worth noting that Evolutionary Algorithms are most accurately thought of as high-level frameworks for solving problems, rather than complete and ready-to-use solutions. An EA expert will thus design variation operators, the fitness function and other components on a per-problem basis, and combine them into a functional algorithm using the outline given above.

The remainder of this section is organized as follows. Section 2.2.1 describes EA components that depend heavily on the problem and/or the particular EA technique being used to solve it. Section 2.2.2 covers components that depend on representation. Finally, section 2.2.3 describes universal EA components. This particular breakdown

is due to [29].

2.2.1 Problem-Specific Components

We now proceed to describe problem-specific components: fitness function and fitness evaluation, and representation. The exact form of these components depends heavily on the problem that they are used to solve.

Fitness Function and Fitness Evaluation

The fitness function, here denoted f, is responsible for providing a numeric quality measure $\mathbf{q}_i$ for candidate solution x_i:

$$f(x_i) = \mathbf{q}_i,\ \mathbf{q}_i \in \mathbb{R}^d$$

where d is the number of objectives in the problem. For example, in the autotuning setting where $d = 2$, the fitness function returns a vector $\mathbf{q}_i$ with two components: runtime and accuracy. In the special case $d = 1$, we call f and the problem it represents *single-objective*. Otherwise, when $d \geq 2$, we call them *multi-objective*. The process of computing the fitness value for a given candidate solution is called *fitness evaluation*, or *evaluation* for short. Even though EAs are effective under a variety of fitness functions (and their geometric interpretation, *fitness landscapes*), an implicit requirement is that the fitness function be mostly continuous. If it is not, EAs tend to act like random search [29].

In some settings, such as program autotuning, the fitness function is *stochastic* or *noisy*. This means that the values of f are samples from some underlying and often unknown probability distribution F. In such cases we assume that there exists one true fitness value equal to the mean of the underlying distribution, and the goal is to approximate it quickly and accurately.

Many EAs use a black-box approach to fitness evaluation, in which the values $f(x_i)$ can be readily computed but little is known about f itself, and no closed form formulation is available. As such, candidate fitness can rarely be predicted ahead of

time without invoking f, which can be problematic if fitness evaluations are costly.

The cost and feasibility of fitness evaluation can be a major factor in the design of an evolutionary algorithm. Some approaches, such as Genetic Algorithms, often rely on a sizable population, whose evaluation at each generation might be prohibitive if f is expensive to compute. In such cases, alternative methods such as Interactive Evolutionary Computation (IEC) are used [46].

Representation

A crucial issue in Evolutionary Computation is *representation*, or the low-level encoding of candidate solutions [37]. Representation provides a bridge from the original problem context to the EA context where the actual optimization takes place [28, 29]. Objects in the original problem space are commonly referred to as *phenotypes*, while their encodings in the EA context are called *genotypes*, Formally, representation specifies a two-directional mapping from phenotypes to their corresponding genotypes [28]. The process of mapping phenotypes to genotypes is called *encoding*, while its inverse is referred to as *decoding*. As an example, consider optimizing an integer-valued function. The algorithm's user might choose binary representation as the encoding, and hence the phenotype 42 would be encoded as 101010. Similarly, the phenotype 011111 would be decoded as 31.

While the distinction between phenotypes and genotypes might seem minor, it is important to understand that EA search happens in the genotype space [28]. The shape and characteristics of that space might be significantly different from those of the phenotype space. To that extent, a good representation encodes candidate solutions in a manner that makes the optimization easier by, for example, creating a smooth search landscape.

Two representations are of particular importance to the autotuners presented in this thesis:

- Integer representation: each candidate solution is encoded as a vector of integers, where each integer is referred to as a *gene*. A particular gene controls one or many (or none, in pathological cases) aspects of the individual's phenotype.

- Tree representation: candidates are encoded as trees, which can represent e.g. decision trees for selecting the appropriate algorithm.
- Hybrid representation: candidate solutions are encoded as a combination of trees and integer vectors.

2.2.2 Representation-Dependent Components

Two EA components: initialization and variation operators, do not depend directly on the problem but on the representation [29]. We proceed to describe them in more detail.

Initialization

Initialization specifies how the initial (first) population gets chosen. In most applications, it is relatively simple: the first population consists of individuals generated at random from some probability distribution [28, 29]. In others, a problem-specific heuristic is used. While in principle such heuristics could be used for any problem, their relative cost and benefits need to be evaluated on a per-problem basis [28].

Variation Operators

The role of *variation operators* is to explore the search space by creating new individuals and thus introducing random variation into the population [28, 29]. We can classify variation operators into two categories: mutation and crossover, based on their arity. Arity in the context of variation operators specifies how many candidates an operator takes as inputs [28].

- **Mutation** is the name given to unary variation operators. It is applied to the genotype of one candidate solution, and outputs a slightly modified genotype commonly called a *child* or *offspring*. The modifications are usually stochastic [28]. The role of mutation varies by EC algorithm - in Evolutionary Programming it is the only operator responsible for search, while in Genetic Programming it is often not used at all [28]. Regardless of the specifics, however, the

general role of mutation is to perform small steps in the search space and ensure that the space is connected, i.e. all the points are reachable given enough time. Connectedness ensures that a global optimum is theoretically obtainable [28].

- **Crossover** or **recombination** is the name given to variation operators of arity at least 2 [29]. Such operators produce offspring using information from at least two parent genotypes, and for that reason are often called *sexual*. The role of crossover is to combine different parts of parent genotypes into a new offspring solution, hoping that the offspring will retain and/or improve the good traits of its parents. Similarly to mutation, crossover is a stochastic process and choosing the parts to combine usually involves randomness. The role of recombination varies by EC algorithms - they are often the only variation operator in Genetic Programming, and an important one in Genetic Algorithms [42, 28]. In contrast, Evolutionary Programming does not use recombination at all [28].

2.2.3 General Components

Despite differences in fitness function, variation operators and representation, surprisingly many components are common to all EC algorithms. These components are outlined below.

Population

A population is a set of genotypes which represent candidate solutions currently under consideration. In a sense, population is the unit of evolution [28], because a standard EC algorithm operates by adapting and improving its population of solutions, rather than any single candidate solution. Given the parent population, an EC applies variation operators to selected individuals and thus produces the offspring population.

There are two important metrics that describe populations: size and diversity. Size is simply the number of individuals within a population, and is usually constant. Choosing the right size is an important aspect of EC algorithm design, as it can affect search time and the algorithm's robustness in noisy fitness settings [5, 18, 8].

Diversity, on the other hand, describes the amount of variation between candidate solutions. Common diversity metrics are the variance among fitness values, or the number of unique genotypes. Entropy is also sometimes used [28].

Parent Selection

Parent selection is the process of selecting parent solutions for use in mutation and recombination. The goal is to select only parents whose offspring have a high chance of improving their parents' fitness. This is usually accomplished through some variant of *fitness-proportionate* selection, i.e. candidates with higher fitness values have a higher chance of being selected for reproduction. Low-quality candidates are selected more rarely, although in many applications they are selected sometimes in order to prevent the search from getting stuck in local optima [28]. Parent selection is generally randomized.

Survivor Selection Mechanism

Survivor selection is similar to parent selection in that its purpose is to distinguish candidates based on their quality. Unlike parent selection, however, it is applied *after* offspring solutions have been generated. Since the population size is usually constant in EC, survivor selection is responsible for deciding which parents and which offspring are allowed into the next population (*next generation*). This process is usually deterministic: a fitness-biased EC might rank both parents and offspring by fitness and select only the first few, bounded by the preset population size. An age-biased EC, on the other hand, might select only from the offspring [28].

Termination Condition

Most EC algorithms have no guarantees about finding the optimum solution in some reasonable bounded time. As such, the algorithm's user has to specify one or more heuristic termination conditions. Some common ones include [28]:

- $\pm\epsilon$ within optimum: the maximum theoretical value of the fitness function

is known, and the search is terminated when it comes to within $\pm\epsilon$ of that optimum.

- time limit: user-defined maximum running time has elapsed. Other related measures, such as CPU time, the number of generations or the number of fitness evaluations can be used as well.
- convergence: the search has converged, i.e. fitness improvement in the last few generations stayed below some small threshold.
- diversity loss: the population diversity drops below a predefined threshold.

In many cases, a combination of the above termination conditions is used. For example, an algorithm might be terminated either when it comes to within $\pm\epsilon$ of optimum, *or* a time limit has passed, whichever comes first [28].

2.2.4 Genetic Algorithms

This section provides background on Genetic Algorithms, a variant of Evolutionary Algorithms implemented by the GPEA (Section 3.1) and thus most relevant to this thesis. Genetic Algorithms are the most widely know type of evolutionary algorithms, initially conceived by Holland as a means of studying adaptive behavior [28].

While there is some variation within genetic algorithms, some authors describe a "classical genetic algorithm" also known as the "simple GA" (SGA) [28]. The simple GA is single-objective (the fitness function returns a single number) and can be easily characterized using the component framework outlined in the previous section (2.2.3).

The simple GA uses bit strings as its genotype representation, and maintains a population of candidates of constant size m [42, 28, 37]. A genotype in GAs is commonly referred to as the *chromosome*, and its length d is fixed. Recombination is achieved through 1-Point bit-wise crossover, where the value of a bit at the given position is replaced with the value of the corresponding bit in the other parent. The crossover operator is applied probabilistically with the *crossover rate* p_c. That is, having selected two parents, an offspring will be created through recombination with

probability p_c. Otherwise, with probability $1 - p_c$, offspring are created asexually by copying the parents.

The most common mutation operator, known as uniform, randomly flips bits in the chromosome [42, 37]. More specifically, given a mutation probability p_m, each bit is independently flipped with probability p_m. Thus for a chromosome of length d, an expected number of $p_m \times d$ bits are flipped.

Some GA variants, such as the GPEA presented in this thesis, use an integer representation with integers instead of single bits. The common variation operators in such a setting are defined analogously - mutation draws a random integer and crossover swaps corresponding integer values.

The simple GA uses fitness-proportional selection as the parent selection mechanism. That is, if the population consists of the candidates $x_1, x_2, \ldots, x_m$, the probability of a candidate x_i becoming a parent is:

$$Pr(x_i \text{ is selected as parent}) = \frac{f(x_i)}{\sum_{j=1}^{m} f(x_j)}$$

Another common selection variant is *tournament selection*. Instead of considering the entire population, tournament selection picks k individuals at random (with or without replacement), and adds the most fit one to the mating pool. This process is repeated until the mating pool has reached the desired size, usually equal to the population size m [28].

Survival selection in SGA is generational: the set of survivors is selected after the offspring have been generated. A common technique is *age-based replacement*, where the offspring completely replace the parents regardless of their fitness. This is the approach taken by the SGA. Another technique is *fitness-based*, where the age is ignored in favor of fitness. Many schemes that combine the two approaches exist. The one important to this thesis is *elitism*, introduced by Kenneth de Jong in 1975. Elitism works like age-based replacement, but the fittest member (or a fixed number of the fittest members) are always carried over to the next generation, regardless of their age [28, 42].

2.2.5 Multi-objective Algorithms

A special class of problems solved by Evolutionary Algorithms are multi-objective problems, i.e. problems where the fitness function f returns a vector with at least 2 components, each component corresponding to a different objective. While in principle multiple objectives could be reduced to a single objective through a weighted sum, in practice such sums may have no easy interpretation. More importantly, the designer might be interested exactly in how one objective might be traded off against another, a notion that a collapsed objective does not capture [37]. In such problems, the job of an EA is to optimize directly with respect to the multiple objectives [28, 37].

In the context of multi-objective optimization, an important notion is that of *Pareto Optimality*. A solution is Pareto Optimal if none of its objectives can be improved without sacrificing at least one other objective [28, 37]. There can exist multiple Pareto Optimal solutions, corresponding to different objective trade-offs. We call the set of such solutions the *Pareto optimal front*, and individual solutions within that front *non-dominated*. Similarly, a solution is *dominated* if there exists another solution whose all objective values are at least as high, and at least one objective is strictly higher.

2.3 The PetaBricks Language

The PetaBricks language provides a framework for the programmer to describe multiple ways of solving a problem while allowing the autotuner to determine which of those ways is best for the user's situation [4]. It provides both algorithmic flexibility (multiple algorithmic choices) as well as coarse-grained code generation flexibility (synthesized outer control flow).

At the highest level, the programmer can specify a *transform*, which takes some number of inputs and produces some number of outputs. In this respect, the PetaBricks transform is like a function call in a procedural language. The major difference is that we allow the programmer to specify multiple pathways to convert the inputs to the outputs for each transform. Pathways are specified in a dataflow manner us-

ing a number of smaller building blocks called *rules*, which encode both the data dependencies of the rule and `C++`-like code that converts the rule's inputs to outputs.

Dependencies are specified by naming the inputs and outputs of each rule, but unlike in a traditional dataflow programming model, more than one rule can be defined to output the same data. Thus, the input dependencies of a rule can be satisfied by the output of one or more rules. It is up to the PetaBricks compiler and autotuner to decide which rules to use to satisfy such dependencies by determining which are most computationally efficient for a given architecture and input. For example, on architectures with multiple processors, the autotuner may find that it is preferable to use rules that minimize the critical path of the transform, while on sequential architectures, rules with the lowest computational complexity may fair better. The following example will help to further illustrate the PetaBricks language.

2.3.1 Example PetaBricks Program: kmeans

Figure 2-3 presents an example PetaBricks program, `kmeans`, that implements the k-means clustering algorithm. The input to the algorithm is a set of n points $x_1, x_2, \ldots, x_n$ (`Points`) and the number of clusters k, $k \leq n$. The algorithm's goal is to find k cluster centers $\mu_1, \mu_2, \ldots, \mu_k$ (`Centroids`) and partition of points between the clusters $S_1, S_2, \ldots, S_k$ (`Assignments`) such that the following error function is minimized [40]:

$$\arg\min_{\mu_1,\ldots,\mu_k} \sum_{i=1}^{k} \sum_{x_j \in S_i} ||x_j - \mu_j||^2$$

Intuitively, the k-means algorithm tries to discover natural clusters in data, where the number k of clusters is specified beforehand by the user. An example run of k-means is shown in Figure 2-2.

The PetaBricks implementation of the k-means algorithm works as follows. It groups the input `Points` into a number of clusters and writes each points cluster to the output `Assignments`. Internally the program uses the intermediate data `Centroids` to keep track of the current center of each cluster. The transform header declares each

of these data structures as its inputs (`Points`), outputs (`Assignments`), and intermediate or "through" data structures (`Centroids`). The rules contained in the body of the transform define the various pathways to construct `Assignments` from `Points`. The transform can be depicted using the dependence graph shown in Figure 2-4, which indicates the dependencies of each of the three rules.

The first two rules specify different ways to initialize the `Centroids` data needed by the iterative solver in the third rule. Both of these rules require the `Points` input data. The third rule specifies how to produce the output `Assignments` using both the input `Points` and intermediate `Centroids`. Note that since the third rule depends on the output of either the first or second rule, the third rule will not be executed until the intermediate data structure `Centroids` has been computed by one of the first two rules. Additionally, the first rule provides an example of how the autotuner can synthesize outer control flow. Instead of explicitly looping over every column of `Centroids` 2D array, the programmer can specify a computation that is done for each column of the output (using the `column` keyword). The order over which these columns are iterated, and the amount of parallelism to use, is then synthesized and tuned by the compiler and autotuner.

To summarize, when our transform is executed, the cluster centroids are initialized either by the first rule, which performs random initialization on a per-column basis with synthesized outer control flow, or the second rule, which calls the `CenterPlus` algorithm. `CenterPlus` implements the `kmeans++` algorithm (not shown), details of which can be found in [9]. Once `Centroids` is generated, the iterative step in the third rule is called.

2.3.2 Variable Accuracy Algorithms

One of the key features of the PetaBricks programming language is support for variable accuracy algorithms, which can trade output accuracy for computational performance (and vice versa) depending on the needs of the user. Approximating ideal program outputs is a common technique used for solving computationally difficult problems, adhering to processing or timing constraints, or optimizing performance in

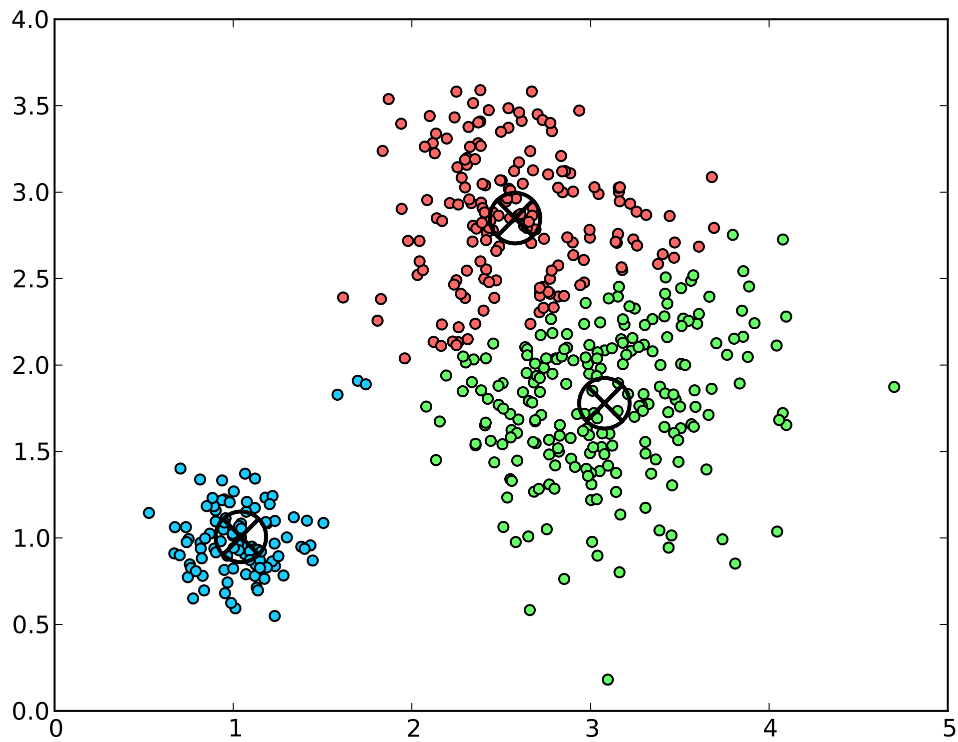


Figure 2-2: An example run of the `kmeans` algorithm on a set of 2-D points (`Points`), with the number of clusters fixed at 3. The crosshairs mark cluster centers (`Centroids`) and different point colors (`Assignments`) correspond to different clusters.

```
1  transform kmeans
2  accuracy_metric kmeansaccuracy // User-supplied accuracy metric
3                                 // (invoked automatically by the
4                                 // autotuner to test the accuracy
5                                 // of a given k-means
6                                 // configuration).
7
8  accuracy_variable k            // Optimal value will be set by
9                                 // the autotuner.
10
11 from Points[n,2]               // Array of points (each column
12                                // stores x and y coordinates).
13 through Centroids[k,2]
14 to Assignments[n]
15 {
16   // [Rule 1] First initialization alternative:
17   // Choose a random point as the initial
18   // mean of each cluster
19   to(Centroids.column(i) c) from(const Points p) {
20     c = p.column(rand(0,n))
21   }
22
23   // [Rule 2] Second initialization alternative:
24   // use the kmeans++ approximation algorithm
25   // (implemented by CenterPlus, see [9] for details)
26   to(Centroids c) from(const Points p) {
27     c = CenterPlus(p);
28   }
29
30   // [Rule 3] Iterative refinement
31   to(Assignments a) from(const Points p, Centroids c) {
32     for_enough {
33       int change;
34       (change, a) = AssignClusters(a, p, c);
35       if (change==0) return; // Reached fixed point
36       c = NewClusterLocations(c, p, a);
37     }
38   }
39 }
```

Figure 2-3: PetaBricks pseudocode for `kmeans`

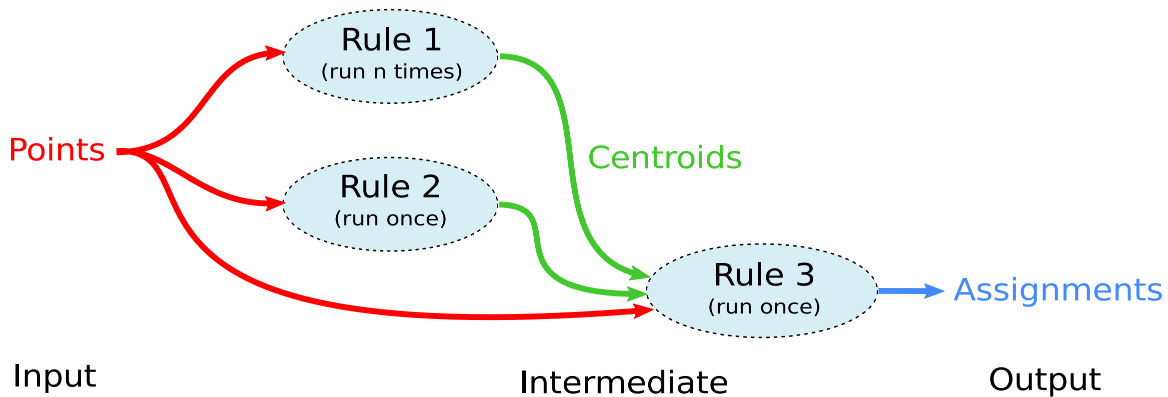


Figure 2-4: Dependency graph for `kmeans` example. The rules are the vertices while each edge represents the dependencies of each rule. Each edge color corresponds to each named data dependence in the pseudocode.

situations where perfect precision is not necessary. Algorithmic methods for producing variable accuracy outputs include approximation algorithms, iterative methods, data resampling, and other heuristics. A detailed description of the variable accuracy features of PetaBricks is given in [3].

At a high level, PetaBricks extends the idea of algorithmic choice to include choices between different accuracies. Language extensions allow users to specify how accuracy should be measured for their transforms. The autotuner simultaneously optimizes for both performance and accuracy, producing a set of optimal algorithms that meet a range of accuracy levels. Users can specify whether they want output accuracy to be met statistically or guaranteed through the use of run-time accuracy checking.

The `kmeans` example presented in Figure 2-3 is a variable accuracy algorithm. We briefly describe the variable accuracy features used in this example. The `accuracy_metric` keyword on line 2 points to the user-defined transform, `kmeansaccuracy`, which computes the accuracy of a given input/output pair to `kmeans`. PetaBricks uses this transform during autotuning (and optionally at runtime) to test the accuracy of a given configuration of `kmeans`. The variable k (lines 8 and 13) controls the number of clusters the algorithm generates by changing the size of the array `Centroids`. Since k can have different optimal values for different input sizes and accuracy levels, declaring k an `accuracy_variable` (line 8) instructs the autotuner to automatically find assignments of this variable during training to satisfy various levels of accuracy.

The `for_enough` loop on line 32 is a loop where the compiler can pick the number of iterations needed for each accuracy level and input size.

During training the autotuner will explore different assignments of k, algorithmic choices of how to initialize the `Centroids`, and iteration counts for the `for_enough` loop to discover efficient algorithms for various levels of accuracy.

2.4 PetaBricks Autotuning

The autotuner must identify selectors that will determine which choice of an algorithm will be used during a program execution so that the program executes as fast as possible while meeting a user-defined QoS target. PetaBricks uses the accuracy of the program as the QoS, and hence that target is called the *accuracy target*. Formally, a selector s consists of $\mathbf{C_s} = [c_{s,1}, \ldots, c_{s,m-1}] \cup \mathbf{A_s} = [\alpha_{s,1}, \ldots, \alpha_{s,m}]$ where $\mathbf{C_s}$ are the ordered interval boundaries (cutoffs) associated with algorithms $\mathbf{A_s}$. During program execution the runtime function SELECT chooses an algorithm depending on the current input size by referencing the selector as follows:

$$SELECT(input, s) = \alpha_{s,i} \text{ s.t. } c_{s,i} > size(input) \geq c_{s,i-1}$$

where

$$c_{s,0} = min(size(input)) \text{ and } c_{s,m} = max(size(input)).$$

The components of $\mathbf{A_s}$ are indices into a discrete set of applicable algorithms available to s, which we denote $Algorithms_s$. The maximum number of intervals is fixed by the PetaBricks compiler. An example of a selector for a sample sorting algorithm is shown in Figure 2-5. In addition to algorithmic choices, the autotuner also tunes user-defined integer parameters (*tunables*) such as accuracy variables (see Section 2.3.1), blocking sizes, sequential/parallel cutoffs and the number of worker threads. Each tuned parameter is thus either an index into a small discrete set or an integer in some positive bounded range.

Formally, given a program P, hardware H and input size n, the autotuner must

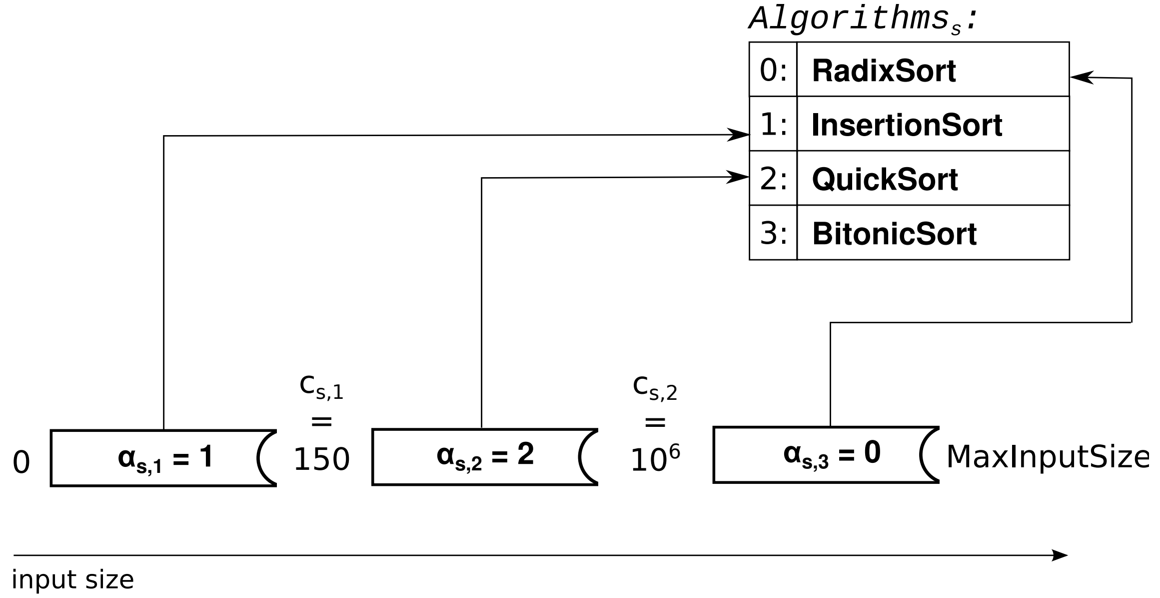


Figure 2-5: A selector for a sample sorting algorithm where $\mathbf{C_s} = [150, 10^6]$ and $\mathbf{A_s} = [1, 2, 0]$. The selector selects the *InsertionSort* algorithm for input sizes in the range $[0; 150)$, *QuickSort* for input sizes in the range $[150, 10^6)$ and *RadixSort* for $[10^6, MAXINT)$. *BitonicSort* was suboptimal for all input ranges and is not used.

identify the vector of selectors $\mathbf{S}$ and vector of tunables $\mathbf{T}$ such that the following objective function ϕ is maximized:

$$\phi(\mathbf{S}, \mathbf{T}, P, H, n) = \begin{cases} runtime(\mathbf{S}, \mathbf{T}, P, H, n)^{-1} & \text{if } acc(\mathbf{S}, \mathbf{T}, P, H, n) \geq acc_{target} \\ 0 & \text{otherwise} \end{cases}$$

In other words, the autotuner attempts to find a set of selectors $\mathbf{S}$ and tunables $\mathbf{T}$ which maximize program throughput (inverse of the running time) while meeting the user-specified target accuracy acc_{target}.

For each compiled program, the PetaBricks compiler produces a binary executable and a configuration file. The configuration file contains selector parameters $\mathbf{A}_s$ and $\mathbf{C}_s$ for each algorithmic choice, as well as the vector of tunables $\mathbf{T} = [t_1, \ldots, t_l]$. Each PetaBricks program contains a randomized generator of sample input data, and can be automatically benchmarked for any given input size n without having to explicitly specify the input itself.

2.4.1 Properties of the Autotuning Problem

Three properties of autotuning influence the design of an autotuner. First, the cost of fitness evaluation depends heavily on the input data size used when testing the candidate solution. The autotuner does not necessarily have to use the target input size. For efficiency it could use smaller sizes to help it find a solution to the target size because is generally true that smaller input sizes are cheaper to test on than larger sizes, though exactly how much cheaper depends on the algorithm. For example, when tuning matrix multiply one would expect testing on a 1024×1024 matrix to be about 8 times more expensive than a 512×512 matrix because the underlying algorithm has $O(n^3)$ performance. While solutions on input sizes smaller than the target size sometimes are different from what they would be when they are evolved on the target input size, it can generally be expected that relative rankings are robust to relatively small changes in input size. This naturally points to "bottom-up" tuning methods that incrementally reuse smaller input size tests or seed them into the initial population for larger input sizes.

Second, in autotuning the fitness of a solution is measured in terms of how long it takes to run. Therefore the cost of fitness evaluation is dependent on the quality of a candidate algorithm. A highly tuned and optimized program will run more quickly than a randomly generated one and it will thus be fitter. This implies that fitness evaluations become cheaper as the overall fitness of the population improves.

Third, significant to autotuning well is recognizing the fact that fitness evaluation is noisy due to details of the parallel micro-architecture being run on and artifacts of concurrent activity in the operating system. The noise can come from many sources, including: caches and branch prediction; races between dependent threads to complete work; operating system artifacts such as scheduling, paging, and I/O; and, finally, other competing load on the system. This leads to a design conflict: an autotuner can run fewer tests, risking incorrectly evaluating relative performance but finishing quickly, or it can run many tests, likely be more accurate but finish too slowly. An appropriate strategy is to run more tests on less expensive (i.e. smaller)

input sizes.

The INCREA exploits incremental structure and handles the noise exemplified in autotuning. We now proceed to describe a INCREA for autotuning.

2.5 Benchmarks

We show results from eight PetaBricks benchmarks to demonstrate the effectiveness of our online autotuning framework. Table 2.2 lists various attributes of each benchmark, including which are variable and which are fixed accuracy. A brief description of our benchmarks follows.

Benchmark name	Variable accuracy	Search space dimensions
Sort	No	33
Eigenproblem	No	35
Matrix Multiply	No	108
LU Factorization	No	140
Bin Packing	Yes	117
Clustering	Yes	91
Helmholtz	Yes	61
Image Compression	Yes	163
Poisson	Yes	64

Table 2.2: Listing of benchmarks and their properties.

2.5.1 Fixed Accuracy

`Sort` recursively sorts an array of integers utilizing various sorting algorithms (insertion, quick, merge, and radix).

`Eigenproblem` computes the eigenvalues and eigenvectors of a symmetric matrix using various numerical algorithms (divide and conquer, QR, and bisection).

`Matrix Multiply` performs multiplication of two dense matrices using various methods including recursive decompositions and *Strassen's algorithm*.

`LU Factorization` performs a factorization of a dense square matrix $A = LU$, commonly used to solve linear systems.

2.5.2 Variable Accuracy

`Bin Packing` is an NP-hard problem that finds an assignment of items to unit sized bins such that the number of bins used is minimized, all bins are within capacity, and every item is assigned to a bin.

`Clustering`, or `kmeans`, is an NP-hard problem that divides a set of data into clusters based on similarity, which is a common technique for statistical data analysis in areas including machine learning, pattern recognition, image segmentation and computational biology.

`Helmholtz` solves the 3D variable-coefficient Helmholtz equation, a partial differential equation that describes physical systems that vary through time and space, such as combustion and wave propagation.

`Image Compression` performs Singular Value Decomposition (SVD) on an $m \times n$ matrix, which is a major component found in some image compression algorithms [52].

`Poisson` solves the 2D Poisson's equation, an elliptic partial differential equation that describes heat transfer, electrostatics, fluid dynamics, and various other engineering problems.

The benchmarks Sort and Matrix Multiply are described in more detail in [4]. The benchmarks Bin Packing, Clustering, Helmholtz Image Compression, and Poisson are described in more detail in [3]. Additionally an extensive study of the Poisson benchmark can be found in [21].

Chapter 3

Offline Autotuning

An off-the-shelf evolutionary algorithm (EA) does not typically take advantage of short cuts based on problem properties and this can sometimes make it impractical because it takes too long to run. A general short cut is to solve a small instance of the problem first then reuse the solution in a compositional manner to solve the large instance which is of interest. Usually solving a small instance is both simpler (because the search space is smaller) and and less expensive (because the evaluation cost is lower). Reusing a sub-solution or using it as a starting point makes finding a solution to a larger instance quicker. This short cut is particularly advantageous if solution evaluation cost starts high and grows proportionally with instance size. It becomes more advantageous if the evaluation result is noisy or highly variable which requires additional evaluation sampling.

This short cut is vulnerable to local optima: a small instance solution might become entrenched in the larger solution but not be part of the global optimum. Or, non-linear effects between variables of the smaller and larger instances may imply the small instance solution is not reusable. However, because EAs are stochastic and population-based they are able to avoid potential local optima arising from small instance solutions and address the potential non-linearity introduced by the newly active variables in the genome.

In this chapter, we describe the EA and associated offline autotuner called INCREA which incorporates into its search strategy the aforementioned short cut through

incremental solving. It solves increasingly larger problem instances by first activating only the variables in its genome relevant to the smallest instance, then extending the active portion of its genome and problem size whenever an instance is solved. It shrinks and grows its population size adaptively to populate a gene pool that focuses on high performing solutions in order to avoid risky, excessively expensive, exploration. It assumes that fitness evaluation is noisy and addresses the noise early when a lot of resampling is less expensive because smaller instances are being solved.

We will exemplify INCREA by solving the problem known as offline autotuning. Offline autotuning arises as a final task of program compilation in PetaBricks, and its goal is to select tunables and algorithmic choices for the program to make it run as fast as possible. Because a program can have varying size inputs, INCREA tunes the program for small input sizes before incrementing them up to the point of the maximum expected input sizes.

3.1 General-Purpose EA (GPEA)

We compare the performance of INCREA to that of an off-the shelf evolutionary algorithm that we call the General-Purpose EA or GPEA. We now proceed to describe GPEA in terms of the standard framework outlined in Section 2.2.

3.1.1 Representation

The GA represents configuration files as fixed-length chromosomes of length $(2m + 1)k + n$, where k is the number of selectors, m the number of interval cutoffs within each selector and n the number of tunables defined for the PetaBricks program. We keep the number of intervals fixed across selectors, but retain the ability to have fewer effective ones by allowing intervals of length 0.

Each gene can assume any value in the range $[0, MAXINT)$, and encodes either a cutoff, an algorithm or a tunable value with a one-to-one correspondence (Figure 3-1). The respective decoding functions, ϕ_c, ϕ_α and ϕ_t are defined as follows:

$$c_{s,i} = \phi_c(c^*_{s,i}) = \left\lfloor \left(\frac{\sum_{j=1}^{i} c^*_{s,j}}{\sum_{j=1}^{m} c^*_{s,j}} \right) MAXINT \right\rfloor$$

$$\alpha_{s,j} = \phi_\alpha(\alpha^*_{s,j}) = \left\lfloor \left(\frac{\alpha^*_{s,j}}{MAXINT} \right) ||Algorithms_s|| \right\rfloor$$

$$t_j = \phi_t(t^*_j) = lo_j + \left\lfloor \left(\frac{t^*_j}{MAXINT} \right) (hi_j - lo_j) \right\rfloor$$

The decoding functions ϕ_α and ϕ_t scale the encoded genes linearly to within their allowed ranges - $[lo_j, hi_j)$ for tunables and $[0, ||Algorithms_s||)$ for algorithms. The cutoff decoder ϕ_c treats the encoded cutoffs as interval lengths, which it normalizes and sums to get the decoded cutoffs. This formulation of ϕ_c also ensures that consecutive cutoffs are non-decreasing.

3.1.2 Initialization

Initial population consists of a fixed number of configuration files generated by randomizing all cutoff, algorithm and tunable values. The values are drawn from the same distributions which are used by the mutation operator.

3.1.3 Fitness Evaluation

We define the fitness of a genome as the inverse of the runtime. The runtime is obtained by timing the PetaBricks program for some input size n, with the decoded genome as its configuration file. The exact value of the input size is specified by the user.

Dealing with Noisy Fitness

Fitness variance is inherent in the autotuning process due to a varying system load. As a result, reported runtimes can be longer than the true ones, and cause inaccurate fitness values. To mitigate this problem, our GA can time genomes multiple times and use the minimum runtime. Since for large inputs evaluations can dominate the

GA's runtime, we perform multiple timings only for small input sizes. The GPEA further deals with fitness noise by maintaining a large (100) population of candidate solutions. For a discussion on how population size can help in noisy settings, see [8, 18].

3.1.4 Variation Operators

The GA relies on single-gene crossover and mutation to generate successive populations of configuration files. An offspring is generated by first crossing-over two parent configuration files, and then possibly mutating the result. The probability of a crossover is fixed at 1, i.e. all offspring are obtained through crossover, but the mutation probability $P(mutation)$ is adjustable.

Crossover

Given two parents, the crossover operator chooses uniformly at random a single gene in both parents. If the gene encodes an algorithm $\alpha_{s,i}$, the offspring is then equal to the first parent with $\alpha_{s,i}$ substituted from the second. If, on the other hand, the gene encodes a cutoff or a tunable, we either perform a similar substitution or choose a random value in between the two parents' gene values, with a probability of the substitution vs. random choice equal $P(substitution)$.

Mutation

After the crossover operator has been applied to two parents to produce an offspring, the offspring is mutated with probability $P(mutation)$. The mutation operator selects, uniformly at random, a single gene from the offspring. The new value for that gene is then drawn from a different probability distribution depending on whether the gene encodes a cutoff, an algorithm or a tunable.

If the gene encodes an algorithm, the new value is drawn from a uniform probability distribution $0 - ||Algorithms_s||$. If it encodes a cutoff, the new value is a power of 2 drawn from a log-uniform distribution, i.e. $2^0, 2^1, \ldots, 2^{W-1}$ are equally likely (W

denotes the word size). However, if the gene is a tunable, then the distribution is chosen depending on the range of allowed values: uniform if the range is smaller than some constant R, i.e. $hi_i - lo_i < R$, and log-uniform otherwise.

3.1.5 Parent and Survivor Selection

Parents are selected using tournament selection with a fixed tournament size. Survivor selection is age-biased with elitism, where 95% of best offspring are carried over to the next generation, while the bottom 5% are replaced by the best candidates from the parent population.

3.1.6 Termination Condition

The GPEA terminates after a hard limit of 100 generations; there are no other termination conditions.

3.2 Bottom-Up EA (INCREA)

3.3 Representation

The INCREA genome, see Figure 3-1, encodes a list of selectors and tunables as integers each in the range $[0, MaxVal)$ where $MaxVal$ is equal to the cardinality of each algorithmic choice's set for algorithms, and equal to $MaxInputSize$ for cutoffs. Each tunable has a $MaxVal$ which is the cardinality of its value set or a bounded integer depending on what it represents.

In order to tune programs of different input sizes the genome represents a solution for maximum input size and throughout the run increases the "active" portion of it starting from the selectors and tunables relevant to the smallest input size. It has length $(2m + 1)k + n$, where k is the number of selectors, m the number of interval cutoffs within each selector and n the number of other tunables defined for the PetaBricks program. As the algorithm progresses the number of "active" cutoff

and algorithm pairs, which we call "choices" for each selector in the genome starts at 1 and then is incremented in step with the algorithm doubling the current input size each generation.

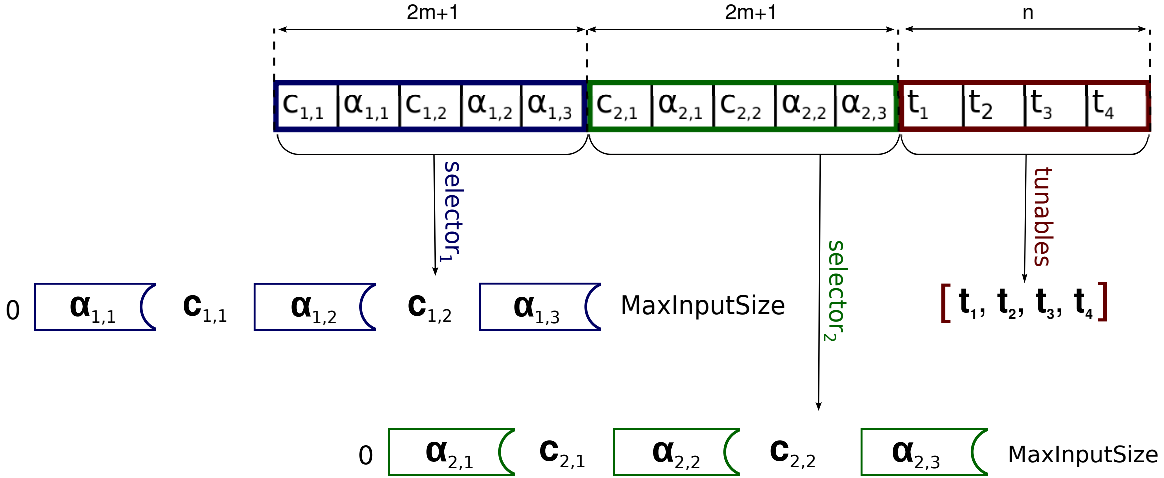


Figure 3-1: A sample genome for $m = 2$, $k = 2$ and $n = 4$. Each gene stores either a cutoff $c_{s,i}$, an algorithm $\alpha_{s,i}$ or a tunable value t_i.

Fitness evaluation

The fitness of a genome is the inverse of the corresponding program's execution time. The execution time is obtained by timing the PetaBricks program for a specified input size.

3.3.1 Top level Strategy

Figure 3-2 shows top level pseudocode for INCREA. The algorithm starts with a "parent" population and an input size of 1 for testing each candidate solution. All choices and tunables are initially set to algorithm 0 and cutoff of MAXINT. The choice set is grown through mutation on a per candidate basis. The input size used for fitness evaluation doubles each generation.

A generation consists of 2 phases: exploration, and downsizing. During exploration, a random parent is used to generate a child via mutation. Only active choices and tunables are mutated in this process. The choice set may be enlarged. The child

is added to the population only if it is determined to be fitter than its parent. The function "fitter" which tests for this condition increases trials of the parent or child to improve confidence in their relative fitnesses. Exploration repeatedly generates a child and tests it against its parent for some fixed number of MutationAttempts or until the population growth reaches some hard limit.

During downsizing, the population, which has potentially grown during exploration, is pruned down to its original size once it is ranked. The "rankThenPrune" function efficiently performs additional fitness tests only as necessary to determine a ranking of which it is reasonably certain.

This strategy is reminiscent but somewhat different from a $(\mu + \lambda)$ES [12]. The $(\mu + \lambda)$ES creates a pool of λ offspring each generation by random draws from the parent population of size μ. Then both offspring and parents are combined and ranked for selection into the next generation. The subtle differences in INCREA are that 1) in a "steady state" manner, INCREA inserts any child which is better than its parent immediately into the population while parents are still being drawn, and 2) a child must be fitter than its parent before it gains entry into the population. The subsequent ranking and pruning of the population matches the selection strategy of $(\mu + \lambda)$ES.

Doubling the input size used for fitness evaluation at each generation allows the algorithm to learn good selectors for smaller ranges before it has to find ones for bigger ranges. It supports subsolution reuse and going forward from a potentially good, non-random starting point. Applying mutation to only the active choice set and tunables while input size is doubling brings additional efficiency because this narrows down the search space while concurrently saving on the cost of fitness evaluations because testing solutions on smaller inputs sizes is cheaper.

3.3.2 Mutation Operators

The mutators perform different operations based on the type of value being mutated. For an algorithmic choice, the new value is drawn from a uniform probability distribution $0 - ||Algorithms_s||$. For an cutoff, the existing value is scaled by a random value

```
1 populationSize = popLowSize
2 inputSizes = [1, 2, 4, 8, 16, ...., maxInputSize]
3 initialize population(maxGenomeLength)
4 for gen = 1 to log(maxInputSize)
5   /* exploration phase: population and active
6      choices may increase */
7   inputSize = inputSizes[gen]
8   for j = 1 to mutationAttempts
9     parent = random draw from population (w/ replacement)
10    activeChoices = getActiveChoices(parent)
11    /* active choices could grow */
12    child = mutate(parent, activeChoices)
13    /* requires fitness evaluations */
14    if fitter(child, parent, inputSize)
15      population = add(population, child)
16      if length(population) >= popHighSize
17        exit exploration phase
18  end /* exploration phase */
19  /* more testing */
20  population = rankThenPrune(population,
21                             popLowSize,
22                             inputSize)
23  /* discard all past fitness evaluations */
24  clearResults(population)
25 end /* generation loop*/
26 return fittest population member
```

Figure 3-2: Top level strategy of INCREA.

drawn from a log-normal distribution, i.e. doubling and halving the existing value are equally likely. The intuition for a log-normal distribution is that small changes have larger effects on small values than large values in autotuning. We have confirmed this intuition experimentally by observing much faster convergence times with this type of scaling.

The INCREA mutation operator is only applied to choices that are in the active choice list for the genome. INCREA has one specialized mutation operator that adds another choice to the active choice list of the genome and sets the cutoff to 0.75 times the current input size while choosing the algorithm randomly. This leaves the behavior for smaller inputs the same, while changing the behavior for the current set of inputs being tested. It also does not allow a new algorithm to be the same as the

one for the next lower cutoff.

3.3.3 Dealing with Noisy Fitness

Because INCREA must also contend with noisy feedback on program execution times, it is bolstered to evaluate candidate solutions multiple times when it is ranking any pair. Because care must be taken not to test too frequently, especially if the input data size is large, it uses an adaptive sampling strategy [1, 19, 20, 47] . The boolean function "fitter", see Figure 3-3, takes care of this concern by running more fitness trials for candidates $s1$ and $s2$ under two criteria. The first criterion is a t-test [41]. When the t-test result has a confidence, i.e. p-value less than 0.05, $s1$ and $s2$ are considered different and trials are halted. If the t-test cannot confirm difference, least squares is used to fit a normal distribution to the percentage difference in the mean execution time of the two algorithms. If this distribution estimates there is a 95% probability of less than a 1% difference, the two candidates' fitnesses are considered to be the same. There is also a parametrized hard upper limit on trials.

The parent ranking before pruning, in function "rankThenPrune", is optimized to minimize the number of additional fitness evaluations. First, it ranks the entire population by mean performance without running any additional trials. It then splits the ranking at the *populationLowSize* element into a $KEEP$ list and a $DISCARD$ list. Next, it sorts the $KEEP$ list by calling the "fitter" function (which may execute more fitness trials). Next, it compares each candidate in the $DISCARD$ list to the *populationLowSize* element in the $KEEP$ list by calling the "fitter" function. If any of these candidates are faster, they are moved to the $KEEP$ list. Finally, the $KEEP$ list is sorted again by calling "fitter" and the first *populationLowSize* candidates are the result of the pruning.

This strategy avoids completely testing the elements of the population that will be discarded. It allocates more testing time to the candidate that will be kept in the population. It also exploits the fact that comparing algorithms with larger differences in performance is cheaper than comparing algorithms with similar performance.

```
1 function fitter(s1, s2, inputSize)
2   while s1.evalCount < evalsLowerLimit
3     evaluateFitness(s1, inputSize)
4   end
5   while s2.evalCount < evalsLowerLimit
6     evaluateFitness(s2, inputSize)
7   end
8   while true
9     /* Single tailed T-test assumes each sample's mean is
        normally distributed.
10       It reports probability that sample means are same under
            this assumption  */
11    if ttest(s1.evalsResults, s2.evalResults) < PvalueLimit /*
        statistically different */
12      return mean(s1.evalResults) > mean(s2.evalResults)
13    end
14    /* Test2Equality: Use least squares to fit a normal
        distribution to the percentage
15       difference in the mean performance of the two algorithms.
             If this
16       distribution estimates there is a 95% probability of less
            than a 1%
17       difference in true means, consider the two algorithms the
            same. */
18    if Test2Equality(s1.evalResults, s2.evalResults)
19      return false
20    end
21    /* need more information, choose s1 or s2 based on the
        highest expected
22       reduction in standard error */
23    whoToTest = mostInformative(s1, s2);
24    if whoToTest == s1 and s1.testCount < evalsUpperLimit
25      evaluateFitness(s1, inputSize)
26    elif s2.testCount < evalsUpperLimit
27      evaluateFitness(s2, inputSize)
28    else
29      /* inconclusive result, no more evals left */
30      return false
31    end
32  end /* while */
33 end /* fitter */
```

Figure 3-3: Pseudocode of function "fitter".

3.4 Experimental Results

3.4.1 Experimental Setup

(a) INCREA

Parameter	Value
confidence required	70%
max trials	5
min trials	1
population high size	10
population low size	2
MutationAttempts	6
standard deviation prior	15%

(b) GPEA

Parameter	Value
mutation rate	0.5
crossover rate	1.0
population size	100
tournament size	10
generations	100
evaluations per candidate	1

Table 3.1: INCREA and GPEA Parameter Settings.

We performed all tests on multiple identical 8-core, dual-Xeon X5460, systems clocked at 3.16 GHz with 8 GB of RAM. The systems were running Debian GNU/Linux 5.0.3 with kernel version 2.6.26. For each test, we chose a target input size large enough to allow parallelism, and small enough to converge on a solution within a reasonable amount of time. Parameters such as the mutation rate, population size and the number of generations were determined experimentally and kept constant between benchmarks. Parameter values we used are listed in Table 3.1.

3.4.2 INCREA vs GPEA

In practice we might choose parameters of either INCREA or GPEA to robustly ensure good autotuning or allow the programmer to vary them while tuning a particular problem and architecture. In the latter case, considering how quickly the tuner converges to the final solution is important. To more extensively compare the two tuners, we ran each tuner 30 times for each benchmark.

Table 3.2 compares the tuners mean performance with 30 runs based on time to convergence and the performance of the final solution. To account for noise, time to convergence is calculated as the first time that a candidate was found that was within

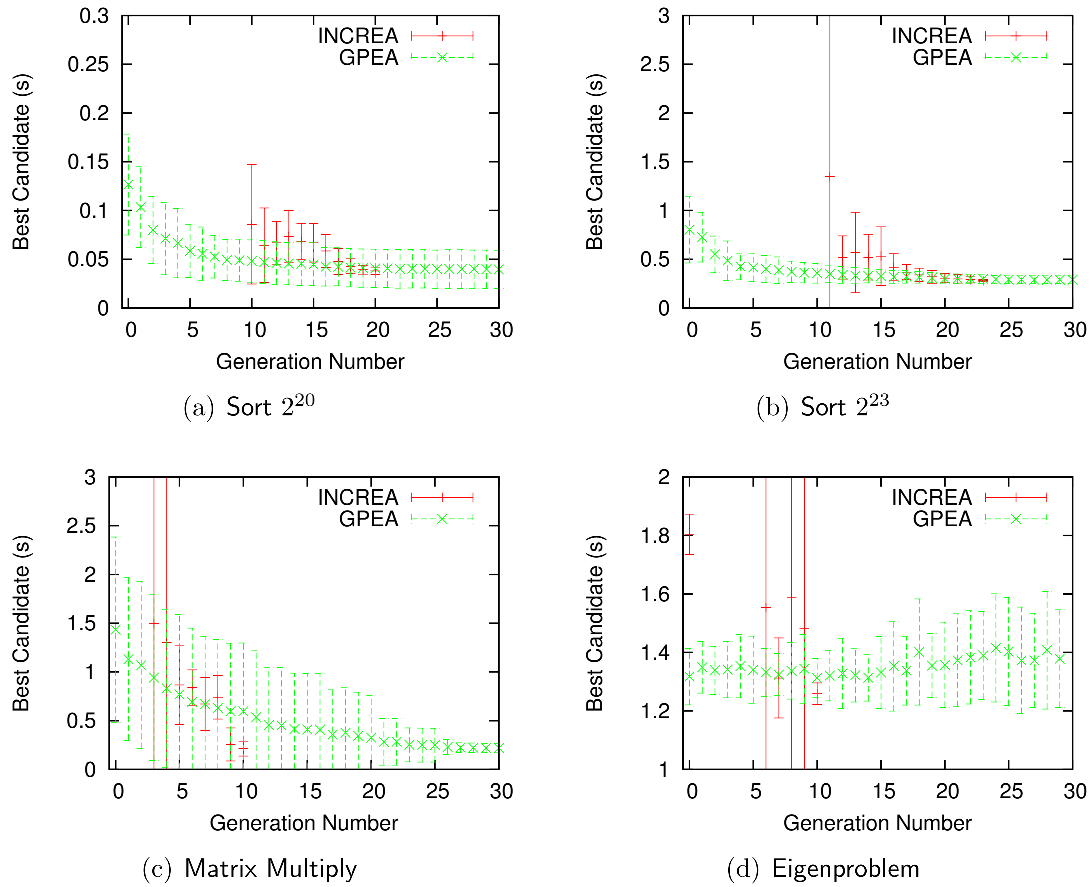
5% of the best fitness achieved. For all of the benchmarks except for Eigenproblem, both tuners arrive a solutions are nearly the same, while for Eigenproblem INCREA finds a slightly better solution. For Eigenproblem and Matrix Multiply, INCREA converges an order of magnitude faster than GPEA. For Sort, GPEA converges faster on the small input size while INCREA converges faster on the larger input size. If one extrapolates convergences times to larger input sizes, it is clear that INCREA scales a lot better than GPEA for Sort.

		INCREA	GPEA	SS?
Sort-2^{20}	Convergence	1464.7 ± 1992.0	599.2 ± 362.9	YES ($p = 0.03$)
	Performance	0.037 ± 0.004	0.034 ± 0.014	NO
Sort-2^{23}	Convergence	2058.2 ± 2850.9	2480.5 ± 1194.5	NO
	Performance	0.275 ± 0.010	0.276 ± 0.041	NO
Matrix Multiply	Convergence	278.5 ± 185.8	2394.2 ± 1931.0	YES ($p = 10^{-16}$)
	Performance	0.204 ± 0.001	0.203 ± 0.001	NO
Eigenproblem	Convergence	92.1 ± 66.4	627.4 ± 530.2	YES ($p = 10^{-15}$)
	Performance	1.240 ± 0.025	1.250 ± 0.014	YES ($p = 0.05$)

Table 3.2: Comparison of INCREA and GPEA in terms of mean time to convergence in seconds and in terms of execution time of the final configuration. Standard deviation is shown after the $\pm$ symbol. The final column is statistical significance determined by a t-test (lower is better).

Figure 3-4 shows aggregate results from 30 runs for both INCREA and GPEA on each benchmark. INCREA generally has a large amount of variance in its early generations, because those generations are based on smaller input sizes that may have different optimal solutions than the largest input size. However, once INCREA reaches its final generation it exhibits lower variance than than GPEA. GPEA tends to converge slowly with gradually decreasing variance. Note that the first few generations for INCREA are not shown because, since it was training on extremely small input sizes, it finds candidates candidates that exceed the timeout set by our testing framework when run on the largest input size. These early generations account for a only a small amount of the total training time.

In 3-4(a) the INCREA's best candidate's execution time displays a "hump" that is caused because it finds optima for smaller input sizes that are not reused in the optimal solution for the target input size.



(a) Sort 2^{20}

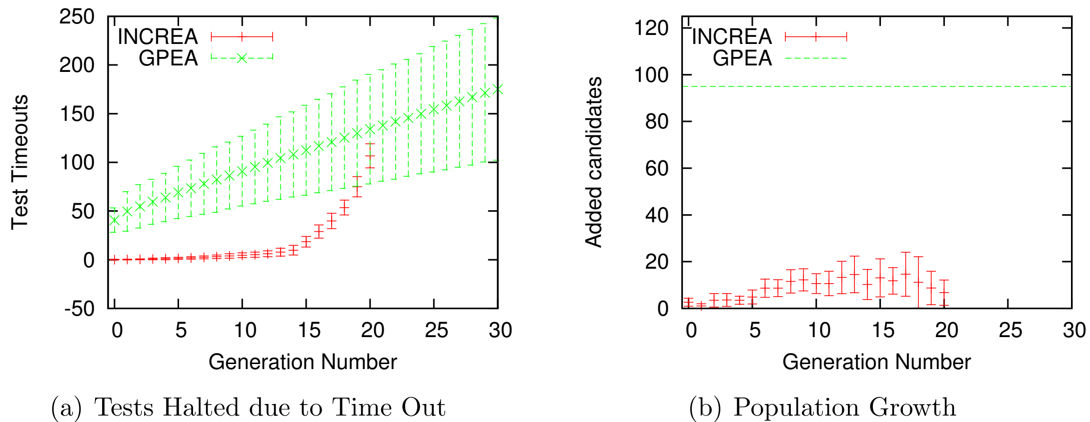
(b) Sort 2^{23}

(c) Matrix Multiply

(d) Eigenproblem

Figure 3-4: Execution time for target input size with best individual of generation. Mean and standard deviation (shown in error bars) with 30 runs.

Using Sort-2^{20}, in Figure 3-5(a) we examine how many tests are halted by each tuner, indicating very poor solutions. The timeout limit for both algorithms is set to be the same factor of the time of the current best solution. However, in GPEA this will always be a test with the target input size whereas with INCREA it is the current input size (which is at least half the time, half as large). Almost half of GPEA's initial population were stopped for timing out, while INCREA experiences most of its timeouts in the later generations where the difference between good and bad solutions grows with the larger input sizes. We also examine in Figure 3-5(b) how much the population grew each generation during the exploration phase. For INCREA the population expansion during exploration is larger in the middle generations as it converges to a final solution.



(a) Tests Halted due to Time Out

(b) Population Growth

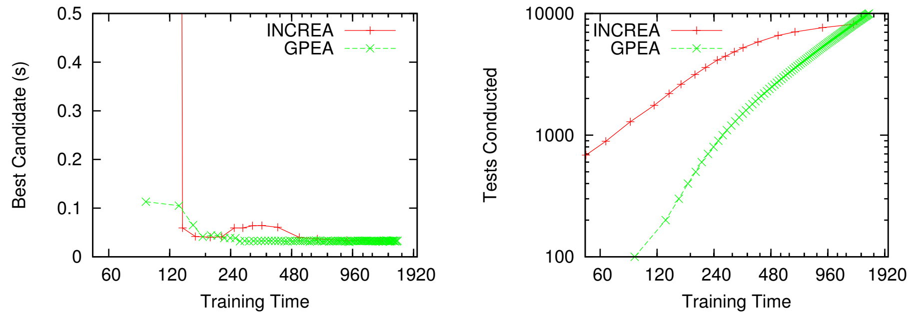
Figure 3-5: Time out and population growth statistics of INCREA for 30 runs of Sort on target input size 2^{20}. Error bars are mean plus and minus one standard deviation.

3.4.3 Representative Runs

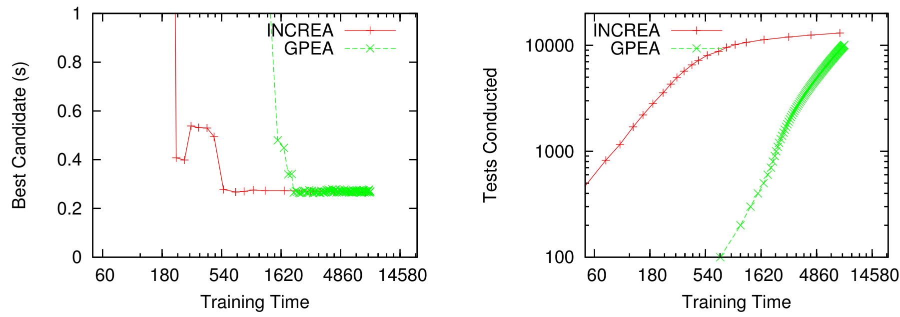
We now select a representative run for each benchmark to focus on run dynamics.

Sort: Sorting

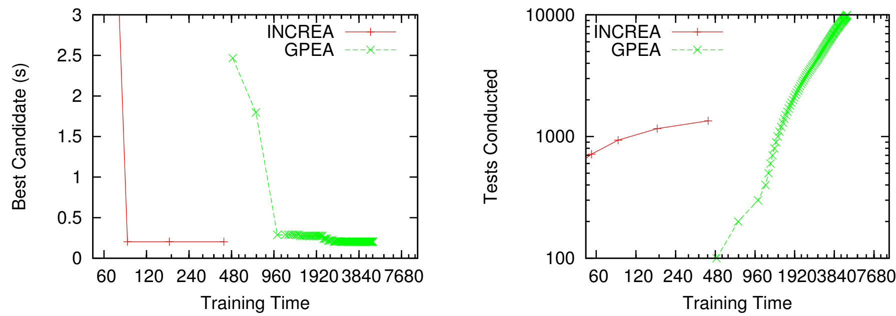
Figures 3-6(a) and 3-6(b) show results from a representative run of each autotuner with two different target input sizes respectively. The benchmark consists of insertion-



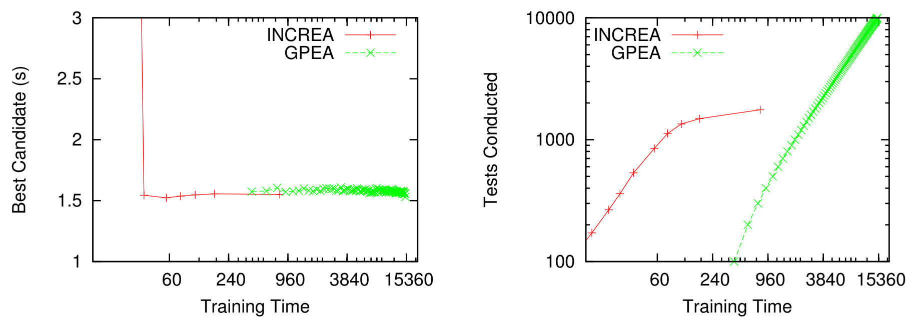
(a) Sort 2^{20}



(b) Sort 2^{23}



(c) Matrix Multiply 1024×1024



(d) Eigenproblem 1024×1024

Figure 3-6: Representative runs of INCREA and GPEA on each benchmark. The left graphs plot the execution time (on the target input size) of the best solution after each generation. The right graph plots the number of fitness evaluations conducted at the end of each generation. All graphs use seconds of training time as the x-axis.

sort, quick-sort, radix sort, and 2/4/8/16/32-way merge-sorts. On this Xeon system, Sort is relatively easy to tune because the optimal solution is relatively simple and the relative costs of the different algorithms are similar.

For the 2^{20} benchmark, both INCREA and GPEA consistently converge to a very similar solution which consists of small variations of quick-sort switching to insertion-sort at somewhere between 256 and 512. Despite arriving at a similar place, the two tuners get there in a very different way. Table 3.3, lists the best algorithm for each tuner at each generation in the run first shown in Figure 3-6(a). INCREA starts with small input sizes, where insertion-sort alone performs well, and for generations 0 to 7 is generating algorithms that primarily use insertion-sort for the sizes being tested. From generations 8 to 16, it creates variants of radix-sort and quicksort that are sequential for the input sizes being tested. In generation 17 it switches to a parallel quick sort and proceeds to optimize the cutoff constants on that for the remaining rounds. The first two of these major phases are locally optimal for the smaller input sizes they are trained on.

GPEA starts with the best of a set of random solutions, which correctly chooses insertion-sort for small input sizes. It then finds, in generation 3, that quick-sort, rather than the initially chosen radix-sort, performs better on large input sizes within the tested range. In generation 6, it refines its solution by parallelizing quick-sort. The remainder of the training time is spent looking for the exact values of the algorithmic cutoffs, which converge to their final values in generation 29.

We classified the possible mutation operations of INCREA and counted how frequently each was used in creating an offspring fitter than its parent. We identified specialized classes of operations that target specific elements of the genome. Table 3.4 lists statistics on each for the run first shown in Figure 3-6(a). The class most likely to generate an improved child scaled both algorithm and parallelism cutoffs. The he class that changed just algorithms were less likely to cause improvement. Overall only 3.7% of mutations improved candidate fitness.

Figure 3-6(c) shows comparative results on Matrix Multiply. The program choices are a naive matrix multiply and five different parallel recursive decompositions, in-

INCREA: Sort		
Input size	Training Time (s)	Genome
2^0	6.9	Q 64 Q_p
2^1	14.6	Q 64 Q_p
2^2	26.6	I
2^3	37.6	I
2^4	50.3	I
2^5	64.1	I
2^6	86.5	I
2^7	115.7	I
2^8	138.6	I 270 R 1310 R_p
2^9	160.4	I 270 Q 1310 Q_p
2^{10}	190.1	I 270 Q 1310 Q_p
2^{11}	216.4	I 270 Q 3343 Q_p
2^{12}	250.0	I 189 R 13190 R_p
2^{13}	275.5	I 189 R 13190 R_p
2^{14}	307.6	I 189 R 17131 R_p
2^{15}	341.9	I 189 R 49718 R_p
2^{16}	409.3	I 189 R 124155 M^2
2^{17}	523.4	I 189 Q 5585 Q_p
2^{18}	642.9	I 189 Q 5585 Q_p
2^{19}	899.8	I 456 Q 5585 Q_p
2^{20}	1313.8	I 456 Q 5585 Q_p

GPEA: Sort		
Gen	Training Time (s)	Genome
0	91.4	I 448 R
1	133.2	I 413 R
2	156.5	I 448 R
3	174.8	I 448 Q
4	192.0	I 448 Q
5	206.8	I 448 Q
6	222.9	I 448 Q 4096 Q_p
7	238.3	I 448 Q 4096 Q_p
8	253.0	I 448 Q 4096 Q_p
9	266.9	I 448 Q 4096 Q_p
10	281.1	I 371 Q 4096 Q_p
11	296.3	I 272 Q 4096 Q_p
12	310.8	I 272 Q 4096 Q_p
...		
27	530.2	I 272 Q 4096 Q_p
28	545.6	I 272 Q 4096 Q_p
29	559.5	I 370 Q 8192 Q_p
30	574.3	I 370 Q 8192 Q_p
...		

Table 3.3: Listing of the best genome of each generation for each autotuner for an example training run. The genomes are encoded as a list of algorithms (represented by letters), separated by the input sizes at which the resulting program will switch between them. The possible algorithms are: I = insertion-sort, Q = quick-sort, R = radix-sort, and M^x = x-way merge-sort. Algorithms may have a $_p$ subscript, which means they are run in parallel with a work stealing scheduler. For clarity, unreachable algorithms present in the genome are not shown.

Mutation Class	Count	Times Tried	Effect on fitness		
			Positive	Negative	None
Make an algorithm active	8	586	2.7%	83.8%	13.5%
Log-normally scale a cutoff	11	1535	4.4%	50.4%	45.1%
Randomly switch an algorithm	12	1343	2.5%	50.4%	25.7%
Log-normally change a parallelism cutoff	2	974	5.2%	38.7%	56.1%

Table 3.4: Effective and ineffective mutations when INCREA solves Sort (target input size 2^{20}.)

cluding Strassen's Algorithm and a cache-oblivious decomposition. A tunable allows both autotuners to transpose combinations of inputs and outputs to the problem. To generate a valid solution, the autotuner must learn to put a base case in the lowest choice of the selector, otherwise it will create an infinite loop. Because many random mutations will create candidate algorithms that never terminate when tested, we impose a time limit on execution.

Both INCREA and GPEA converge to the same solution for Matrix Multiply. This solution consists of transposing the second input and then doing a parallel cache-oblivious recursive decomposition down to 64×64 blocks which are processed sequentially.

While both tuners converge to same solution, INCREA arrives at it much more quickly. This is primarily due to the n^3 complexity of matrix multiply, which makes running small input size tests extremely cheap compared to larger input sizes and the large gap between fast and slow configurations. INCREA converges to the final solution in 88 seconds, using 935 trials, before GPEA has evaluated even 20% of its initial population of 100 trials. INCREA converges to a final solution in 9 generations when the input size has reached 256, while GPEA requires 45 generations at input size 1024. Overall, INCREA converges 32.8 times faster than GPEA for matrix multiply.

Eigenproblem: Symmetric Eigenproblem

Figure 3-6(d) shows results for Eigenproblem. Similar to Matrix Multiply here INCREA performs much better because of the fast growth in cost of running tests. This benchmark is unique in that its timing results have higher variance due to locks and allocation in the underlying libraries used to implement certain mathematical functions. This high variance makes it difficult to autotune well, especially for GPEA which only runs a single test for each candidate algorithm. Both solutions found were of same structure, but INCREA was able to find better cutoffs values than the GPEA.

3.5 Conclusions

INCREA is an evolutionary algorithm that is efficiently designed for problems which are suited to incremental shortcuts and require them because of they have large search spaces and expensive solution evaluation. It also efficiently handles problems which have noisy candidate solution quality. In the so called “real world”, problems of this sort abound. A general purpose evolutionary algorithm ignores the incremental structure that exists in these problems and, while it may identify a solution, it wastes computation, takes too long and produces error prone results. INCREA solves smaller to larger problem instances as generations progress and it expands and shrinks its genome and population each generation. For further efficiency, it cuts off work that doesn’t appear to promise a fruitful result. It addresses noisy solution quality efficiently by focusing on resolving it for small solutions which have a much lower cost of evaluation.

We have demonstrated and evaluated INCREA by solving the offline PetaBricks autotuning problem for multi-scale architectures. INCREA automatically determines user-defined tunables and algorithmic choices that result in the fastest execution for the given hardware. We found that INCREA and a general purpose EA both achieve significant speedups on 3 benchmarks but INCREA is much more efficient because of its exploitation of the problem’s bottom up structure and its greedy efficiencies.

Chapter 4

Online Autotuning

While offline autotuning provides great performance gains, it has two major problems. First, it adds an additional step to the software installation and upgrade process. Second, offline autotuning is unable to construct programs that respond to dynamically changing conditions. As we will show, changes to machine load can substantially degrade an application's performance. When such changes occur, an offline autotuned algorithm may no longer be the best choice. This situation is further exacerbated in the emerging cloud and data center computing environments, where in addition to sharing a machine with varying load, applications may be transparently migrated between machines, and thus potentially between microarchitectures. Such changes to computer architectures and microarchitectures have been shown to lead to significant performance loss for autotuned applications [4].

In response to some of these challenges, there is a growing body of work [35, 34, 38, 15, 22, 13] focused on creating applications that can monitor and automatically tune themselves to optimize a particular objective (e.g. meeting response time goals by trading quality of service (QoS) for increased performance or lower power usage). In order to provide stability, convergence and predictability guarantees, many of these systems construct (either by hand, or automatically) a linear model of their application and employ control theory techniques to perform dynamic tuning. The success of such techniques depends on the degree to which the configurable choices can be mapped to a linear system, a task that can be difficult when tuning large

complex applications with interdependent configuration choices.

In contrast, offline evolutionary (a.k.a. genetic) autotuning techniques, such as the one used in PetaBricks [4, 5], are model-free. They adaptively sample the search space of candidate solutions and take advantage of both large and small moves in the search space. Thus, they are able to find global optima regardless of how non-linear or interdependent the choice space and can do so without a model. Their selection component allows them to improve execution time even though they generate random variations on current solutions with unpredictable performance. When a specific variation is extremely slow, in an offline setting, it can be killed and prevent a sampling bottleneck. Unfortunately, this characteristic has meant that evolutionary techniques are generally considered to be unsuitable for use in the online setting. Executing multiple generations of a sizable population at runtime (even at periodic intervals) is too costly to be feasible. Additionally, the alternative approach of continuously replacing the components being executed by different experimental variations, offered by an autotuner, is also a poor choice as there is no execution standard to compare the variation against. Thus, the learning system has no way of knowing whether a particular variation is performing particularly poorly and thus should be aborted.

In this thesis, we take a novel approach to online learning that enables the application of evolutionary tuning techniques to online autotuning. Our technique, called *SiblingRivalry*, divides the available processor resources in half and runs the current best algorithm on one half and a variation on the other half. If the current best finishes first, the variation is killed, the failure of the variation is reported to the online learning algorithm which controls the selection of both configurations for such "competitions" and the application continues to the next stage. If the variation finishes first, we have found a better solution than the current best. Thus, the current best is killed and the results from the variation are used as the program continues to the next stage. Using this technique, SiblingRivalry produces predictable and stable executions, while still exploiting an evolutionary tuning approach. The online learning algorithm is capable of adapting to changes in the environment and progressively identifies better configurations over time without resorting to experiments that might

deliver extremely slow performance. As we will show, despite the loss of resources, this technique can produce speedups over fixed configurations when the dynamic execution environment changes. To the best of our knowledge, SiblingRivalry is the first attempt at employing evolutionary tuning techniques to online autotuning computer programs.

We have implemented a prototype of the SiblingRivalry algorithm within the context of the PetaBricks language [4, 3]. Our results show that SiblingRivalry's always-on racing technique can lead to an autotuned algorithm that uses only half the machine resources (as the other half is used for learning) but that is often faster and more energy efficient than an optimized algorithm that uses the entire processing resources of the machine. Furthermore, we show that SiblingRivalry dynamically responds and adapts to changes in the runtime environment such as system load.

4.1 Competition Execution Model

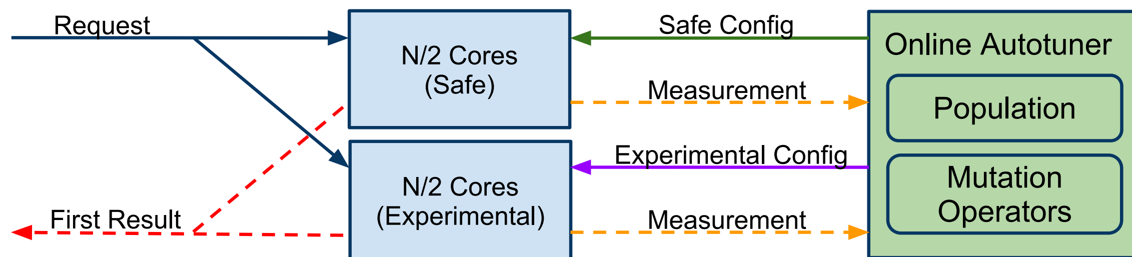


Figure 4-1: High level flow of the runtime system. The data on dotted lines may not be transmitted for the slower configuration, which can be terminated before completion.

Figure 4-1 shows the high level flow of how requests are processed by the PetaBricks runtime system. The cores on our system are split in half into two groups. One group of cores is designated to run safe configurations, while the other group runs experimental configurations. When a request is received, the autotuner runs the same request on both groups of cores in parallel using a safe configuration on one group and an experimental configuration on the other group. When the first configuration completes (and provides a satisfactory answer) the system terminates the slower one. The output of the better algorithm is returned to the user, and timing and quality of

service measurements are sent to the autotuner so that it may update its population of configurations and mutation operator priorities.

4.1.1 Other Splitting Strategies

Our racing execution model requires that there be two groups of cores, one that executes an experimental configuration, while the other executes a safe configuration. While we have chosen to divide our resources in a 50/50 split, other divisions (such as 60/40 or 75/25) are possible.

We do not consider splits where we devote fewer cores to the experimental group than the safe group since doing so would prevent some superior configurations from completing (they would be killed immediately after the safe strategy completes). Further, tuning for fewer than half of the cores limits the potential benefits from autotuning.

One of the reasons we chose a 50/50 split over other possible splits was to minimize the gap between best-case and worst-case overheads that result from splitting. Splits that devote very few resources to the safe configuration will incur extremely large costs when the experimental configuration fails compared to when it succeeds.

Another major advantage of the 50/50 split is that it provides more data to the autotuner, since it receives measurements from two configurations, with at least lower-bound data, per request. In uneven configurations, very little is learned about the configuration on the smaller part of the chip, since even if it is a better configuration it still may be aborted before completion. This means that the online learner is expected to converge more quickly in the 50/50 case.

4.1.2 Time Multiplexing Races

Another racing strategy is to run the experimental configuration and the safe configuration in sequence rather than in parallel. This allows both algorithms to utilize the entire machine. It also provides a way to, in some cases, avoid running the safe configuration entirely. These types of techniques are also the most amenable to at

some point switching off online learning, if one knows that the dynamic execution environment has stabilized and the learner has converged.

There are two variants to this type of technique:

- *Safe configuration first.* In this variant, the safe configuration is run first, and is always allowed to complete, using the entire machine. Then the experimental configuration is allowed an equal amount of time to run, to see if it would have completed faster. Unfortunately, this method will incur a $2x$ overhead in the steady state, which is in practice far more than the overhead of running the races in parallel. For this reason this technique is only desirable if one plans to disable online learning part way through an execution.

- *Experimental configuration first.* In this variant, a model is required to predict the performance of a configuration given a specific input and current dynamic system environment. The model predicts the upper bound performance of the safe configuration. The experimental configuration is given this predicted amount of time to produce an answer before being terminated. If the experimental configuration produces an acceptable answer, then the safe configuration is never run, otherwise the system falls back to the safe configuration.

  The efficacy of this technique depends a lot on the quality of the model used and the probability of the learning system producing bad configurations. In the best case, this technique can have close to zero overhead. However, in the worst case, this technique could both fail to converge and produce overheads exceeding 2x. If the performance model under-predicts execution time, superior configurations will be terminated prematurely and autotuning will fail to make improvements. If the performance model over-predicts execution time, then the cost of exploring bad configurations will grow. For our problem, the probability of a bad configuration is high enough that this type of technique is not desirable, however, with search spaces with more safer configurations this technique may become more appealing.

4.2 SiblingRivalry

The online learner is an evolutionary algorithm (EA) that is specially designed for the purpose of identifying, online, the best configuration for the program. It has a multitude of exacting requirements: It must be lightweight because it is always running. It cannot add significant computational or memory overhead to the application or it will diminish the overall value of autotuning. It must conduct its search in accordance with the structure of the pairwise competition execution model as described in Section 4.1. Accordingly, it must effectively search and adapt candidate solutions by offering competition configurations and integrating the feedback from their measurement results. Because the competition execution model is processing real requests, it must provide at least one configuration that is sufficiently safe to ensure quality of service. Despite the search space of candidate configurations being very large, it must converge to a high quality configuration quickly. It must not assume the underlying environment is stationary. It must converge in the face of high execution time variability (due to load variance) and react to system changes in a timely way without being notified of them.

To meet its convergence goals, the online learner, in effect, must ideally balance exploration and exploitation in its search strategy. Exploitation should investigate candidates in the "neighborhood" of currently high performing configurations. Exploration should investigate candidates that are very different from the current population to ensure no route to the optimum has been overlooked by the greedy nature of exploitation. This final required property of the online learner motivates one of its key capabilities. The online learner performs "adaptive mutator selection" which we explain in more detail in Section 4.2.4.

4.2.1 High Level Function

In the process of tuning a program, the online learner maintains a population of candidate configurations. The population is relatively small to minimize the computational and memory overhead of learning.

The online learner keeps two types of performance logs: per-configuration and per-mutator. Per-configuration logs record runtime, accuracy, and confidence for a given candidate, and are used by the learner to select the "safe" configuration for each competition, and to prune configurations which are demonstrably suboptimal. Per-mutator logs record performance along the three objectives for candidates generated by a given mutator. This information allows the online learner to select mutators which have a record of producing improved solutions, using a process called Adaptive Mutator Selection (see Section 4.2.4 for more information).

Whenever the program being tuned receives a request, the online learner selects two configurations to handle it: "safe" and "experimental". The safe configuration is the configuration with the highest value of the fitness function (see Section 4.2.2) in the current population, computed using per-configuration logs. The fitness value captures how well the configuration has performed in the past, and thus the safe configuration represents the best candidate found by the online learner so far. The experimental configuration is produced by drawing a "seed" configuration from the current population and transforming it using a mutator. The probability of a configuration being selected as a seed is proportional to its fitness.

Once the safe and experimental configurations have been selected, the online learner uses both to process the request in parallel, and returns the result from the candidate that finishes first and meets the accuracy target (the "winner"). The slower candidate (the "loser") is terminated. If the experimental configuration is the winner, it is added to the online learner's population. Otherwise, it is discarded. The safe configuration is added back to the population regardless of the result of the race, but might be pruned later if the new result makes it worse than any other candidate. Details are provided in the pseudocode of Figure 4-2.

The online learner optimizes three objectives with respect to its candidate configurations:

- *Execution time:* the expected execution time of the algorithm.

- *Accuracy:* the expected value of a programmer metric measuring the quality

```
1 population = [initial_configuration]
2 mutators = [operator1, operator2, ...]
3 mutation_weights_time = [1.0, 1.0, ...]
4 mutation_weights_accuracy =  [1.0, 1.0, ...]
5 performance_history = ...
6
7 while True:
8
9   if meeting_accuracy_targets(performance_history):
10     mutation_operator = select_mutator(mutators,
         mutation_weights_time)
11   else
12     mutation_operator = select_mutator(mutators,
         mutation_weights_accuracy)
13
14   safe_cfg = select_safe_config(population, peformance_history)
15   seed_cfg = select_seed_config(population, peformance_history)
16   experimental_cfg = apply_mutator(seed_config, mutation_operator
       )
17
18   request = wait_for_request()
19   result, measurements = compete(request, safe_cfg,
       experimental_cfg)
20   respond_to_request(request, result)
21
22   record_measurements(measurements, performance_history, safe_cfg
       , experimental_cfg)
23   update_mutation_weights(measurements, mutation_operator,
       mutation_weights_time, mutation_weights_accuracy)
24   if not measurements.experimental_cfg_aborted:
25     population.add(experimental_cfg)
26     population.prune(performance_history)
```

Figure 4-2: Pseudocode of how requests are processed by the online learning system

of the solution found.

- *Confidence:* a metric representing the online learner's confidence in the first two metrics. This metric is 0 if there is only one sample and

$$Confidence = \frac{1}{stderr(\text{timings})} + \frac{1}{stderr(\text{accuracies})}$$

  if there are multiple samples. This takes into account any observed variance in the objective. If the observed variance were constant, the metric would be proportional to $sqrt(T)$ where T is the number of times the candidate has been used.

Confidence is an objective because we expect the variance in the execution times and accuracies of a configuration (as it performs more and more competitions) to be significant. Confidence allows configurations with reliable performance to be differentiated from those with highly variable performance. It prevents an "outlier run" from making a suboptimal configuration temporarily dominate better configurations and forcing them out of the population.

Taken together, these objectives create a 3-dimensional space in which each candidate algorithm in the population occupies a point. In this 3-dimensional space, the online learner's goal is to push the current population towards the Pareto optimal front.

4.2.2 Selecting the Safe and Seed Configuration

Each configuration of the population is assigned a fitness, m, that is updated every time it competes against another configuration. Fitness depends upon how well the configuration is meeting a target accuracy, m_a, and its execution time, m_t:

$$m_{config} = \begin{cases} \frac{-m_t}{\sum_{n \in P} n_t} - z\frac{g - m_a}{\sum_{n \in P} n_a} & \text{if } m_a < g \\ \frac{-m_t}{\sum_{n \in P} n_t} & \text{if } m_a \geq g \end{cases}$$

where g is a target accuracy and z is scalar weight set based on how often the

online learner has been meeting its goals in the past. Fitness prioritizes meeting the accuracy target, but gives no reward for accuracy exceeding the target.

To select the safe configuration, the online learner picks the algorithm in the population that has the highest fitness. When the online learner is not producing configurations that meet the targets, the weight of z is adaptively incremented to put more importance on accuracy targets when it calculates m.

To select a seed configuration, the online learner first eliminates any configuration that has an expected running time that is below the 65th percentile running time of the safe configuration. Then, it randomly draws a configuration from the remaining population using the fitness of each configuration to weight the draw. In evolutionary algorithm terminology, this type of draw is called "fitness-proportional selection".

4.2.3 Mutation Operators

The online learner changes configurations of candidate algorithms though a pool of mutation operators that are generated automatically from information outputted by the PetaBricks compiler. Mutators create a new algorithm configuration from an existing configuration by randomly making changes to a specific target region of the configuration.

One can divide the mutators used by our online learner into the following categories:

- **Decision tree manipulation mutators** randomly either add, remove, or change levels of a decision tree. A decision tree is an abstract hierarchically ordered representation of the selector parameters in the configuration file. It enables the dynamic determination of which algorithm to use at a specific dynamic point in program execution. Each level of the tree has a cutoff value and an algorithmic choice. Each decision tree in the configuration results in 5 mutation operators: one operator to add a level, one operator to remove a level, one operator to make large random changes, and two operators to make small random changes.

- **Log-normal random scaling mutators** scale a configuration value by a ran-

dom number taken from a log-normal distribution with scale of 1. This type of mutator is used to change cutoff values that are compared to data sizes. For example, blocking sizes, cutoffs in decision trees, and cutoffs to switch between sequential and parallel code.

- **Uniform random mutators** replace an existing configuration value with a new value taken from a discrete uniform random distribution containing all legal values for the configuration item. This type of mutator is used for choices where there are a relatively small number of possibilities. An example of this is deciding the scheduling strategy for a specific block of code or algorithmic choices.

- **Function manipulation mutators** change the underlying parameter of a function that is used to decide a value that must change dynamically based on input size. For example, the number of iterations in a `for_enough` loop. These functions are represented by $\lg n$ points in the configuration value with runtime interpolation to find values lying between the specified points.

4.2.4 Adaptive Mutator Selection (AMS)

The evolutionary algorithm of the online learner uses different mutators. This provides it with flexibility to generate experimental configurations that range from being close to the seed configuration to far from it, thus controlling its exploration and exploitation. However, the efficiency of the search process is sensitive to *which* mutators are applied and *when*. These decisions cannot be hard-coded because they are dependent on what program is being autotuned. Furthermore, even for a specific program, they might need to change over the course of racing history as the population changes and converges. Mutators that cause larger seed-experiment configuration differences should be favored in early competitions to explore while ones that cause smaller differences should be favored when the search is close to the best configuration to exploit.

For this reason, the online tuner has a specific strategy for selecting mutators

on the basis of how well they have performed. The performance of mutators is the extent to which they have generated experimental configurations of better fitness than others. In general, this is called "Adaptive Operator Selection" (AOS) [25, 48, 24] and our version is called "Adaptive Mutator Selection" (AMS).

There are two parts to AMS: credit assignment to a mutator, and mutator selection. AMS uses *Fitness-based Area-Under-Curve* for its credit assignment and a *Bandit* decision process for mutator selection. We use *Fitness-based Area-Under-Curve* because it is appropriate for the comparison (racing) approach taken by the online learner. We use the AUC version of the *Dynamic Multi-Armed Bandit* decision process because it matches up with the online learner's dynamic environment. Our descriptions are adapted and implemented directly from [30].

4.2.5 Credit Assignment

After each competition the AOS stops and assigns credit to operators based on their performance over a time interval. *Fitness-based Area-Under-Curve* adapts the Area Under the ROC Curve criteria [17] to assign credit to comparison-based assessment of mutators by first creating a ranked list of the experimental configurations generated in any time window according to a fitness objective. The ROC (Receiver Operator Curve) associated to a given mutator, μ, is then drawn by scanning the ordered list, starting from the origin: a vertical segment is drawn when the current configuration has been generated by μ, a horizontal segment is drawn otherwise, and a diagonal one is drawn in case of ties (see Figure 4-3, reproduced from [29]). Finally, the credit assigned to mutator μ is the area under this curve (AUC).

4.2.6 Bandit Mutator Selection

The bandit-based mutator selection deterministically selects the mutator based on a variant of the Upper Confidence Bound (UCB) algorithm [10]:

$$\text{Select } \arg\max_i \left(AUC_{i,t} + C\sqrt{\frac{2\log\sum_k n_{k,t}}{n_{i,k}}} \right) \tag{4.1}$$

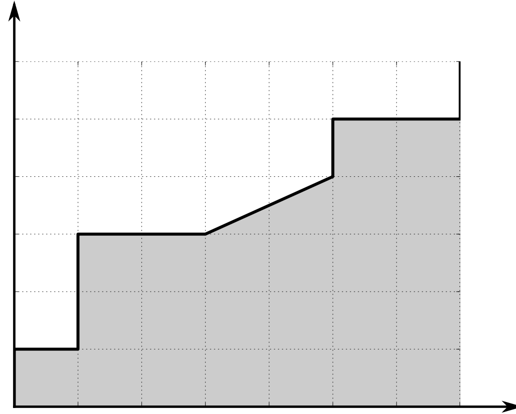


Figure 4-3: The credit assigned to mutator μ is the area under the curve. Section 4.2.5 provides details. Reproduced from [29].

where $AUC_{i,t}$ denotes the empirical quality of the i-th mutator during a user-defined time-window W (exploitation term), $n_{i,t}$ the total number of times it has been used since the beginning of the process (the right term corresponding to the exploration term), and C is a user defined constant that controls the balance between exploration and exploitation. We collectively refer to the constants C and W as *hyperparameters*. For the rest of this chapter, we assume hyperparameters are fixed by the user at $C = 1$ and $W = 50$, and provide an in-depth analysis of their influence in Chapter 5. Bandit algorithms have been proven to optimally solve the exploration vs. exploitation dilemma in a stationary context. The dynamic context is addressed in this formulation by using AUC as the exploitation term. See [45] for more details.

4.2.7 Population Pruning

Each time the population has an experimental configuration added, it is pruned. Pruning is a means of ensuring the experimental configuration should appropriately stay in the population and removing any configuration wholly inferior to the experimental configuration. The experimental configuration should stay if, for any weighting of its objectives, it is better than any other configuration under the same weighting.

This condition is expressed as:

$$\arg\max_{m \in P}\left(\frac{w_a}{\sum_{n \in P} n_a} m_a - \frac{w_t}{\sum_{n \in P} n_t} m_t + \frac{w_c}{\sum_{n \in P} n_c} m_c\right)$$

where P is the population and w defines a weight. The subscripts $_a$, $_t$, and $_c$ of w represent the accuracy, time, and confidence objectives for each configuration.

If the experimental configuration results in an extant configuration no longer being non-dominated, the extant configuration is pruned. We set $w_t = 1 - w_a$ and sample values of w_a and w_c in the range $[0, 1]$. We sample the time-accuracy trade-off space more densely than the confidence space, with approximately 100 different weight combinations total.

4.3 Related Work

A number of offline empirical autotuning frameworks have been developed for building efficient, portable libraries in specific domains. PHiPAC [16] is an autotuning system for dense matrix multiply, generating portable C code and search scripts to tune for specific systems. ATLAS [53, 54] utilizes empirical autotuning to produce a cache-contained matrix multiply, which is then used in larger matrix computations in BLAS and LAPACK. FFTW [31, 32] uses empirical autotuning to combine solvers for FFTs. Other autotuning systems include SPARSITY [36] for sparse matrix computations, SPIRAL [44] for digital signal processing, UHFFT [2] for FFT on multicore systems, OSKI [51] for sparse matrix kernels, and autotuning frameworks for optimizing sequential [39] and parallel [43] sorting algorithms.

In the dynamic autotuning space, there have been a number of systems developed [35, 34, 38, 15, 22, 13] that focus on creating applications that can monitor and automatically tune themselves to optimize a particular objective. Many of these systems employ a control systems based autotuner that operates on a linear model of the application being tuned. For example, PowerDial [35] converts static configuration parameters that already exist in a program into dynamic knobs that can be tuned at

runtime, with the goal of trading QoS guarantees for meeting performance and power usage goals. The system uses an offline learning stage to construct a linear model of the choice configuration space which can be subsequently tuned using a linear control system. The system employs the heartbeat framework [33] to provide feedback to the control system. A similar technique is employed in [34], where a simpler heuristic-based controller dynamically adjusts the degree of loop perforation performed on a target application to trade QoS for performance.

Our work also bears similarities to the Green system [13], whose primary goal is to lower the power requirements of programs. Green uses pragma-like annotations to allow multiple versions of a function that have different power requirements and resulting accuracies. Green uses a global quality of service metric to monitor the impact of running the various approximate versions of the code. PetaBricks differs from Green in that it supports multiple accuracy metrics per program, allows the definition of a much larger class of algorithmic choices and has parallelism integrated with its choice model. We also employ a robust genetic autotuner which does not need a detailed model of the program being tuned, thus avoiding the expensive exploration of the search space. Furthermore, through the use of competitions, our autotuner can effectively deal with highly non-linear configuration spaces without the risk of significantly degrading QoS at any time.

Additionally, there has been a large amount of work [11, 27, 49, 50] in the dynamic optimization space, where information available at runtime is used combined with static compilation techniques to generate higher performing code. Such dynamic optimizations differ from dynamic autotuning because each of the optimizations is hand crafted in a way that makes it likely that it will lead to an improvement in performance when applied. Conversely, autotuning searches the space of many available program variations without a priori knowledge of which configurations will perform better.

Acronym	Processor Type	Operating System	Processors
Xeon8	Intel Xeon X5460 3.16GHz	Debian 5.0	2 (×4 cores)
Xeon32	Intel Xeon X7560 2.27GHz	Ubuntu 10.4	4 (×8 cores)
AMD48	AMD Opteron 6168 1.9GHz	Debian 5.0	4 (×12 cores)

Table 4.1: Specifications of the test systems used and the acronyms used to differentiate them in results.

4.4 Experimental Results

We evaluate SiblingRivalry with two experimental scenarios. In the first scenario, we use a single system and vary the load on the system. In the second scenario we vary the underling architecture, to represent the effects of a computation being migrated between machines. In both cases we compare to a fixed configuration found with offline tuning that utilizes all cores of the underlying machine.

4.4.1 Experimental Setup

We performed our experiments on three systems described in Table 4.1. We refer to these three systems using the acronyms Xeon8, Xeon32, and AMD48. Power measurements were performed on the AMD48 system, using a WattsUp device that samples and stores the consumed power at 1 second intervals.

4.4.2 Sources of Speedups

The speedups achieved for different benchmarks can come from a variety of sources. Some of these sources of speedup can apply even to the case where the environment does not change dynamically. Different benchmarks obtained speedups for different reasons in different tests.

- Algorithmic improvements are a large source of speedup, and the motivation for this work. When the dynamic environment changes, the optimal algorithmic choices may be different and SiblingRivalry can discover better algorithms dynamically.

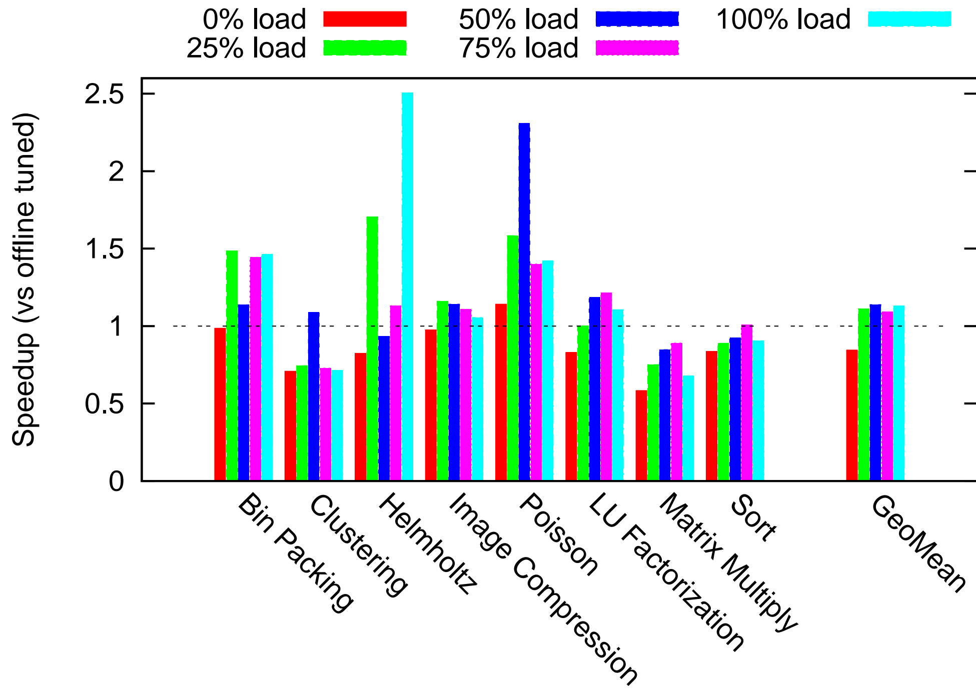
- For the variable accuracy benchmarks, additional speedup can be obtained since the online tuner receives runtime feedback on how well it is meeting its accuracy targets. If it observes that it is over-delivering on its quality of service target it can opportunistically change algorithms, enabling it to be less conservative than offline tuning. For all tests, both SiblingRivalry and the baseline met the required quality of service requirements.

- SiblingRivalry benefits from a "dice effect," since it is running two copies of the algorithm it has an increased chance of getting lucky and having one configuration complete faster than its mean performance. External events, like I/O interrupts, have a lower chance of affecting both algorithms. This leads to a small speedup, which is a function of the variance in the performance of each algorithm.

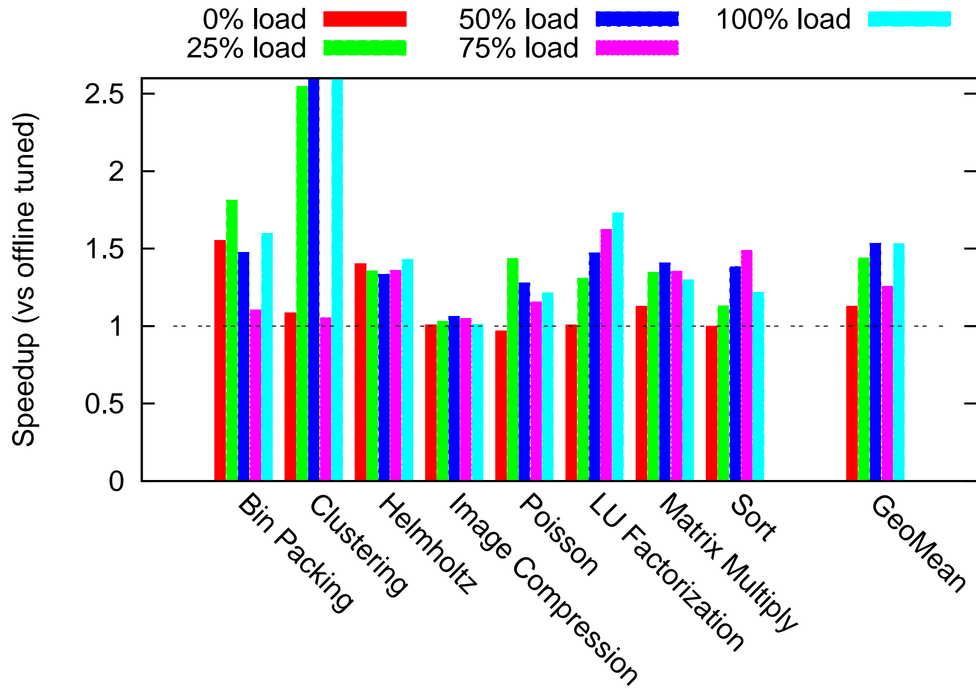
- As the number of processing cores continues to grow exponentially, the amount of per core memory bandwidth is decreasing dramatically since per-chip memory bandwidth is growing only at a linear rate [14]. This fact, coupled with Amdahl's law, makes it particularly difficult to write applications with scalable performance. On our AMD48 machine, we found that some benchmarks with high degree of available parallelism exhibit limited scalability, preventing them from fully utilizing all available processors. In cases where the performance leveled off before half of the available processors, the cost of our competition strategy becomes close to zero.

4.4.3 Load on a System

To test how SiblingRivalry adapts to load on the system, we simulated system load by running concurrently with a synthetic CPU-bound benchmark competing for system resources. We allowed the operating system to assign cores to this benchmark and did not bind it to specific cores. For the different tests, we varied the number of threads in this benchmark to utilize between 0 and 100% of the processors on the system. Combined with the PetaBricks benchmarks, this creates an overloaded system where

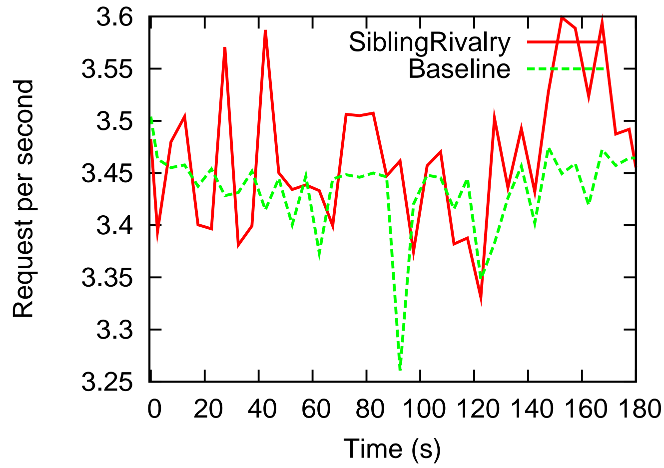


(a) Xeon8

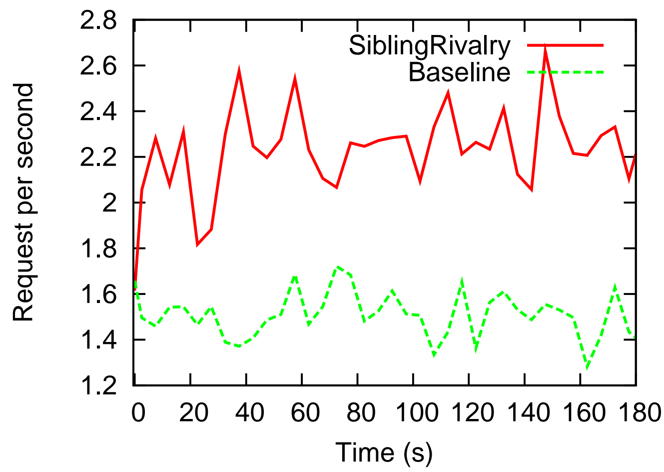


(b) AMD48

Figure 4-4: Speedups (or slowdowns) of each benchmark as the load on a system changes. Note that the 50% load and 100% load speedups for Clustering in (b), which were cut off due to the scale, are 4.0x and 3.9x.



(a) 0% load



(b) 50% load

Figure 4-5: Representative graphs for varying system load showing throughput over time. Benchmark is LU Factorization on AMD48.

the number of active threads is double the number of cores. In all cases we compared SiblingRivalry to a baseline of a fixed configuration found with offline tuning on the same machine, without the additional load. We measure average throughput over 10 minutes of execution, which includes all of the learning costs.

We observed different trends of speedups on the two machines tested. On the Xeon8 (Figure 4-4(a)), the geometric mean cost of running SiblingRivalry (under zero new load) was 16%. This cost is largest for Matrix Multiply, which scales linearly on this system. For other benchmarks, the overheads are lower for two reasons. For the non-variable accuracy benchmarks, some benchmarks do not scale perfectly (These benchmarks exhibit an average speedup of 5.4x when running with 8 threads [4]). For variable accuracy benchmarks, the online autotuner is able to improve performance by taking advantage of using a number of candidate algorithms to construct an aggregate QoS that is closer to the target accuracy level than would be otherwise possible with a single algorithm.

Figure 4-4(b) shows the performance results on the AMD48 machine. In the zero load case, SiblingRivalry achieves a geometric mean speedup of 1.12x. This speedup comes primarily because of the way the autotuner can dynamically adapt the variable accuracy benchmarks (the same way it did on Xeon8). Additionally, while AMD48 and Xeon8 have very similar memory systems, AMD48 has six times as many cores, and thus 6 times less bandwidth per core. Thus, we found that in some cases, using additional cores on this system did not always translate to better performance. For example, while some fixed configurations of our matrix multiply benchmark scale well to 48 cores, our autotuner is able to find a less scalable configuration that provides the same performance using only 20 cores. Once load is introduced, SiblingRivalry is able to further adapt the benchmarks, providing geometric speedups of up to 1.53x.

Figure 4-5 shows an example execution comparing how SiblingRivalry adapts to load on the system. In the zero load case, both system provide roughly the same throughput, with SiblingRivalry being somewhat more noisy. In the 50% load case, both systems initially drop to about 40% of the zero-load throughput. SiblingRivalry recovers quickly (within the first 10 generations) from this initial drop, and settles at

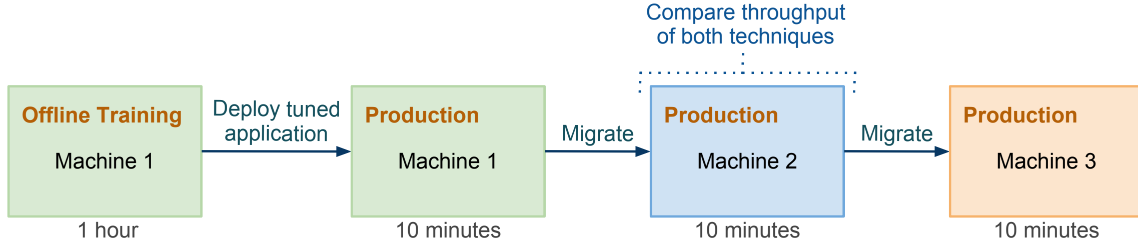


Figure 4-6: The scenario with frequent migration modeled by our architecture migration experiments. We compare a fixed configuration (found with offline training on a different machine) to SiblingRivalry, to show how adapting to each architecture can improve throughput. We measure throughput only between the first and second migration, and include the cost of all learning in the throughput measurements.

around 60% of the original throughput. In contrast, the offline tuned baseline does not recover. The other load levels show similar patterns as the 50% load case.

4.4.4 Migrating Between Microarchitectures

In a second group of experiments we test how SiblingRivalry can adapt to changes in microarchitecture. Figure 4-6 shows the scenario modeled by our experiments. We first train offline on a initial machine and then move this trained configuration to a different machine. We compare SiblingRivalry to a baseline configuration found with offline tuning on the original machine. The offline configuration is given one thread per core on the system. Figure 4-7 shows the speedups for each benchmark after such a migration. SiblingRivalry shows a geometric mean speedup of 1.8x in this migration experiment.

Figures 4-8 and 4-9 expand on Figure 4-7 by showing a number of representative executions in cases where the autotuner made significant changes over the execution. The figures graph how throughput (and accuracy in Figure 4-9) change over time for both SiblingRivalry and the baseline algorithm. The results show how SiblingRivalry is able to make changes to the benchmark configurations to dynamically adapt them over time to improve their throughputs.

Additionally, Figure 4-9(a) shows an example of where the SiblingRivalry dynamically switches to lower accuracy algorithms that are closer to the quality of service

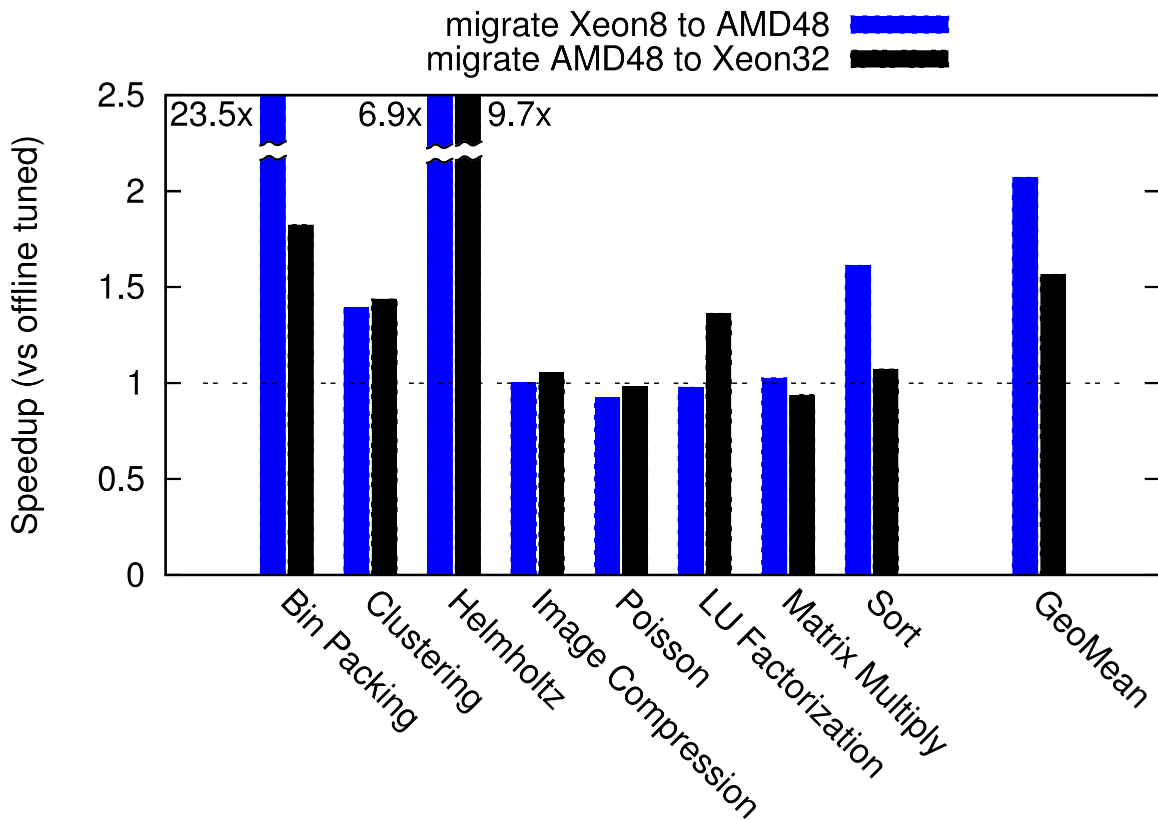
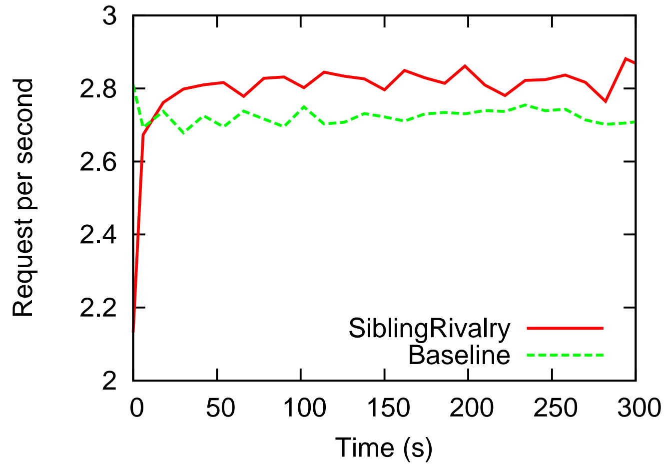
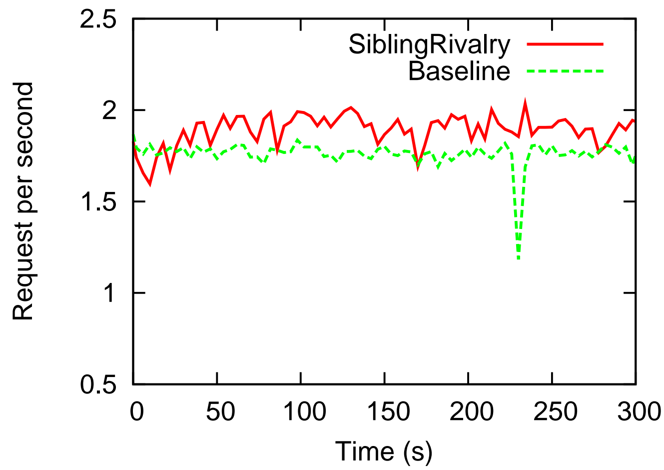


Figure 4-7: Speedups (or slowdowns) of each benchmark after a migration between microarchitectures. "Normalized throughput" is the throughput over the first 10 minutes of execution of SiblingRivalry (including time to learn), divided by the throughput of the first 10 minutes of an offline tuned configuration using the entire system.

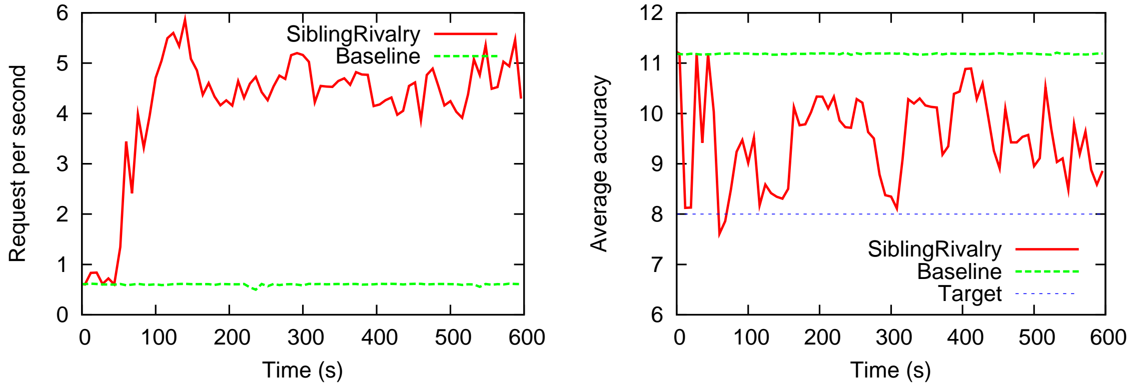


(a) Matrix Multiply, migrate Xeon8 to AMD48

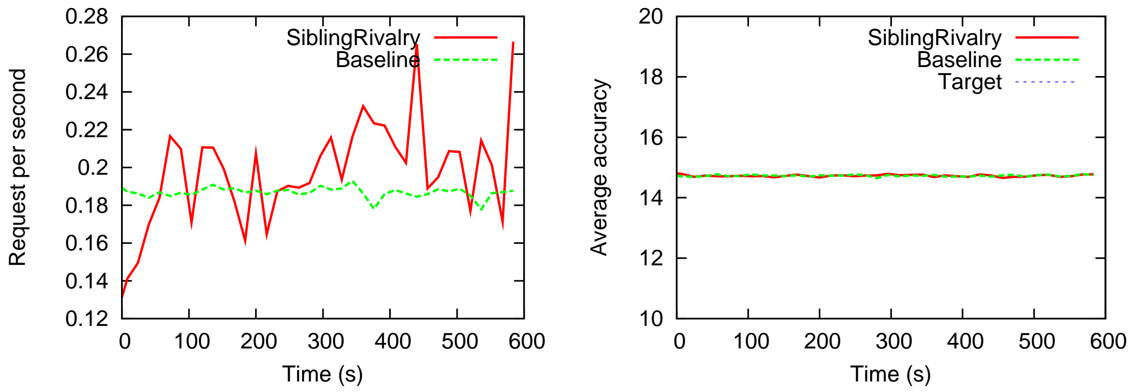


(b) Sort, migrate AMD48 to Xeon32

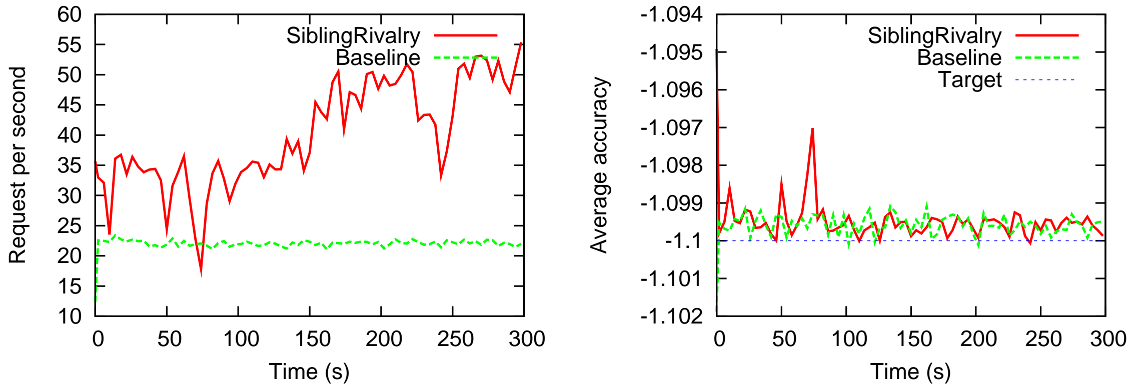
Figure 4-8: Representative graphs of throughput over time for fixed accuracy benchmarks after a migration between microarchitectures.



(a) Helmholtz, migrate Xeon8 to AMD48



(b) Matrix Approximation, migrate AMD48 to Xeon32



(c) Bin Packing, migrate AMD48 to Xeon32

Figure 4-9: Representative graphs of throughput over time for variable-accuracy benchmarks after a migration between microarchitectures. “Target” is the accuracy target both the offline and online tuners are set to optimize for.

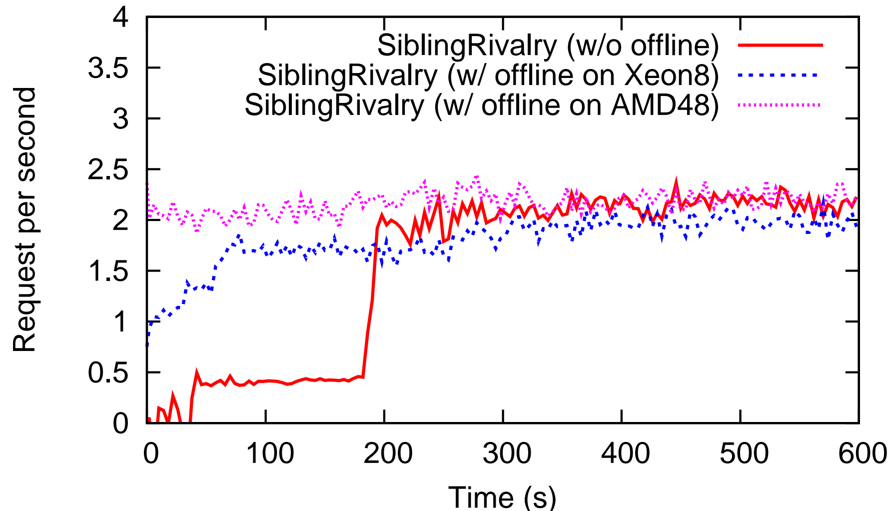


Figure 4-10: The effect of using an offline tuned configuration as a starting point for SiblingRivalry on the Sort benchmark. We compare starting from a random configuration ("w/o offline") to configurations found through offline training on the same and a different architecture.

target. Since the online learner can notice when it is failing to achieve target accuracy and dynamically adjust itself, it can operate closer to the accuracy targets safely. Two other runs exhibit similar behavior: Bin Packing (migrate Xeon8 to AMD48) and Image Compression (migrate Xeon8 to AMD48). Average accuracy did not change by a significant amount for any other benchmark.

4.4.5 Cold Start

Figure 4-10 shows how using an offline tuned configuration affects the rate of convergence of SiblingRivalry. We show three starting configurations: a random configuration, a configuration tuned on a different machine, and a configuration tuned on the same machine. As one would expect, convergence time increases as the starting point becomes less optimal. Convergence times are roughly 5 minutes, 1 minute, and 0 for the configurations tried, though since changes are constantly being made it is difficult to mark a point of convergence.

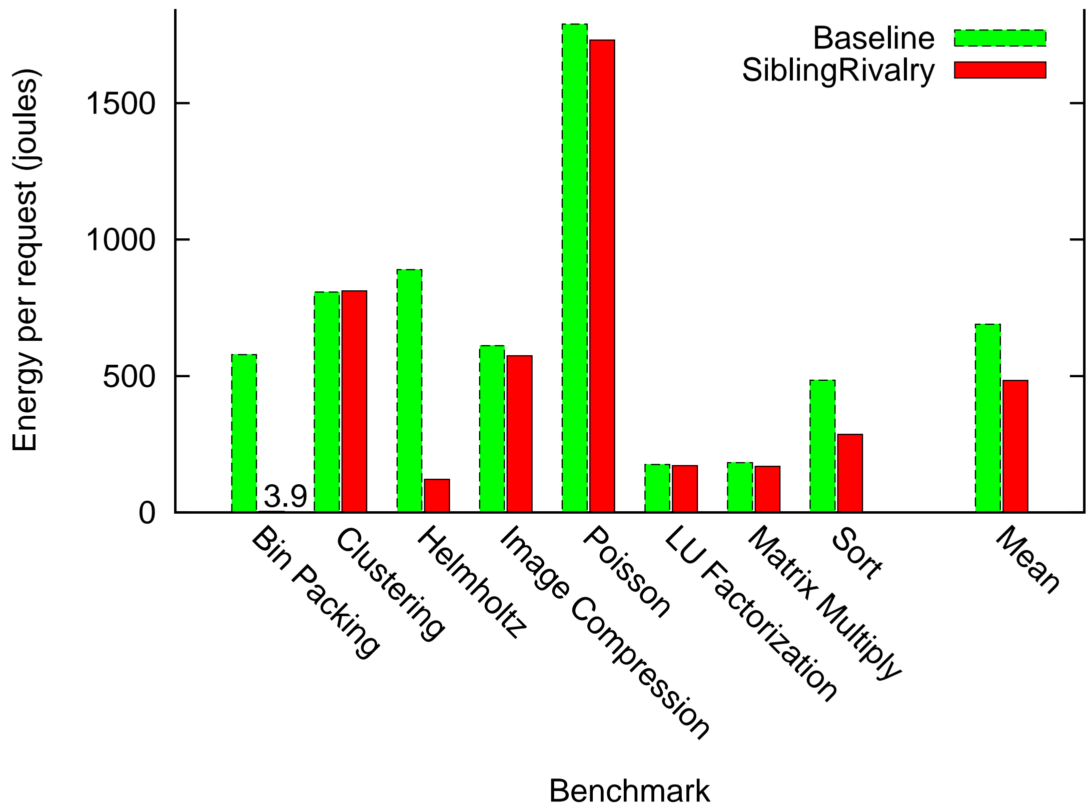


Figure 4-11: Average energy use per request for each benchmark after migrate Xeon8 to AMD48.

4.4.6 Power Consumption

Figure 4-11 shows the energy used per request for each of our benchmarks. While one might initially think that the techniques proposed would increase energy usage since up to twice the amount of work is performed, SiblingRivalry actually reduces energy usage by an average of 30% for our benchmarks. The primary reason for this decreased energy usage is the increased throughput of SiblingRivalry, which results in the machine being used for a shorter period of time. The benchmarks that saw increased throughput also saw decreased power consumption per request.

4.4.7 Conclusions

We demonstrated that it can sometimes be more effective to devote resources to learning the smart thing to do, than to simple throw resources at a potentially suboptimal configuration. Our technique devotes half of the system resources to trying something different, to enable online adaption to the system environment. The geometric mean speedup of SiblingRivalry was 1.8x after a migration between microarchitectures. Even in comparison to an offline-optimized version on the same microarchitecture that uses the full resources, SiblingRivalry showed a geometric mean performance increase of 1.3x when moderate load was introduced on the machine. SiblingRivalry, while performing close to twice the amount of work, consumed on average 30% less power compared to running a well tuned algorithm after a migration. These results show that continuously adapting the program to the environment can provide a huge boost in performance that easily overcame the cost of splitting the available resources in half.

In addition, we have showed that an intelligent machine learning system can rapidly find a good solution even when the search space is extremely large. Furthermore, we demonstrated that it is important to provide many algorithmic and optimization choices to the online learner as done by the PetaBricks language and compiler. While these choices increase the search space, they make it possible for the autotuner to obtain the performance gains observed.

SiblingRivalry is able to fully eliminate the offline learning step, making the process fully transparent to users, which is the biggest impediment to the acceptance of autotuning. For example, while Feedback Directed Optimization (FDO) can provide substantial performance gains, the extra step involved in the programmers workflow has stopped this promising technique from being widely adopted [23]. By eliminating any extra steps, we believe that SiblingRivalry can bring autotuning to the mainstream program optimization. As we keep increasing the core counts of our processors, autotuning via SiblingRivalry help exploit them in a purposeful way.

Chapter 5

Hyperparameter Tuning

The behavior and efficacy of SiblingRivalry is intimately tied to the selection rule in Equation 4.1, which in turn relies on the hyperparameters C and W. It is likely that different values of these hyperparameters can have a significant impact on the quality of autotuned programs, and it is therefore important that we use the appropriate hyperparameters for each tuned program. In the previous chapter, we delegated the problem of selecting such appropriate hyperparameter values to the user, without investigating how difficult they are to find or what impact they have on autotuning. In this chapter, we provide such analysis, which proceeds as follows. In Section 5.1 we discuss the difficulties of selecting optimal hyperparameter values. In Section 5.2, we describe the evaluation metrics we use to formally assess the quality of a given set of a hyperparameter values. Finally, Section 5.3 provides experimental evaluation of SiblingRivalry's hyperparameter robustness.

5.1 Tuning the Tuner

The hyperparameters C (exploration/exploitation trade-off) and W (window size) can have a significant impact on the efficacy of SiblingRivalry. For example, if C is set too high, it might dominate the exploitation term and all operators will be applied approximately uniformly, regardless of their past performance. If, on the other hand, C is set too low, it will be dominated by the exploitation term $AUC_{i,t}$ and new,

possibly better operators will rarely be applied in favor of operators which made only marginal improvements in the past.

The problem is further complicated by the fact that the relative magnitude of the exploration and exploitation terms is highly problem-dependent [29]. For example, programs with a lot of algorithmic choices are likely to benefit from a relatively high exploration rate. This is because algorithmic changes create discontinuities in the program's fitness, and operator scores calculated for a given set of algorithms will not be accurate when those algorithms suddenly change. When such changes occur, exploration should become the dominant behavior. For other programs, e.g. those where only a few mutators improve performance, sacrificing exploration in favor of exploitation might be optimal. This is especially true for programs with few algorithmic choices - once the optimal algorithmic choices have been made, the autotuner should focus on adjusting cutoffs and tunables using an exploitative strategy with a comparatively low C.

The optimal value of C is also closely tied to the optimal value of W, which controls the size of the history window. The autotuner looks at operator applications in the past W races, and uses the outcome of those applications to assign a quality score to each operator. This is based on the assumption that an operator's past performance is a predictor of its future performance, which may not always be true. For example, changes in algorithms can create discontinuities in the fitness landscape, making past operator performance largely irrelevant. However, if W is large, this past performance will still be taken into account for quite some time. In such situations, a small W might be preferred.

Furthermore, optimal values of C and W are not independent. Due to the way $AUC_{i,t}$ is computed, the value of the exploitation term grows with W (see Section 4.2.5). Thus by changing W, which superficially controls only the size of the history window, one might accidentally alter the exploration/exploitation balance. For this reason, C and W should be tuned together.

Finally, the task of selecting hyperparameters is complicated by the fact that different hyperparameter values might be optimal at different stages of the autotun-

ing process. As described earlier, a larger C might be favorable following algorithm changes, with a smaller C when optimal algorithmic choices have already been made. Currently, however, SiblingRivalry does not allow dynamically adjusting hyperparameters throughout the run, which have to be statically set before the autotuning begins.

5.2 Hyperparameter Quality

There is no single metric that will suffice to evaluate SiblingRivalry's performance under different hyperparameter values. For example, if we only look at the average throughput during some fixed time window, we might miss the fact that some initially low-throughput configurations can find optimal solutions but take longer to converge. Likewise, the quality of the best solution might not be very informative if we do not know how long it takes to find that solution, especially if the execution system is dynamic and requires frequent adaptation. For these reasons, we propose five metrics to evaluate SiblingRivalry on a given benchmark program with different hyperparameters:

1. **Mean throughput**: the number of requests processed per second, averaged over the entire duration of the run. Equal to the average number of races per second.

2. **Best candidate throughput**: inverse of the runtime of the fastest candidate found during the duration of the run. For variable accuracy benchmarks, only candidates that met the accuracy target are considered.

3. **Accuracy Root Mean Square Error (RMSE)**: square root of the mean squared distance between the current best candidate's accuracy and the accuracy target. Applies only to variable accuracy benchmarks.

4. **Fraction of races that met accuracy target**: percentage of races that met (to within $\pm 5\%$) or exceeded the accuracy target. Applies only to variable accuracy benchmarks.

5. **Time to convergence**: number of races until a candidate has been found that has a throughput within 5% of the best candidate for the given run. For variable accuracy benchmarks, only candidates that met the accuracy target are considered.

Furthermore, we consider two different evaluation scenarios: static and dynamic system, which correspond to common autotuning setups. In order to enable a fair comparison between SiblingRivalry's performance under different hyperparameter values, we define a single objective metric for each scenario that combines one or more of the five metrics outlined above. We call this metric the *score function* f_b for each benchmark b, and its output the *score*. A detailed description of the static and dynamic scenarios follows.

5.2.1 Static System

In the Static System scenario, the execution system is mostly unchanging - there are no migrations between architectures and the load varies marginally. This scenario corresponds to dedicated hardware that executes the user's program. In this setting, the user cares mostly about the quality of the best candidate. Convergence time is of little concern, as the autotuner only has to learn once and then adapt very infrequently. For the sake of comparison, let's assume in this scenario the user assigns a weight of 80% to the best candidate's throughput, and only 20% to the convergence time. Hence the score function for the static system:

$$f_b(C, W) = 0.8 \times \text{best_throughput}_b(C, W) + 0.2 \times \text{convergence_time}_b^{-1}(C, W)$$

5.2.2 Dynamic System

In the Dynamic System scenario, the user's program might migrate between architectures and/or the load changes considerably and frequently. This scenario corresponds to a shared cloud system with other programs competing for resources. In this setting, the user cares both about average throughput and the convergence time. The

convergence time is a major consideration since execution conditions change often in a dynamic system and necessitate frequent adaptation. Ideally, the autotuner would converge very quickly to a very fast configuration. However, the user is willing sacrifice some of the speed for improved convergence time. We can capture this notion using the following score function:

$$f_b(C, W) = 0.5 \times \text{mean_throughput}_b(C, W) + 0.5 \times \text{convergence_time}_b^{-1}(C, W)$$

While the weights in the score functions might seem arbitrary, they flexibly express the different priorities of the static and dynamic autotuning scenarios. A given set of weights also enables us to find the best-performing values of C and W for each benchmark, and hence provide a way to check how close a particular set of hyperparameters is to that goal. Furthermore, note that if convergence times, mean throughputs and best throughputs are normalized with respect to their best value, then the scores will assume a value in the range $[0; 1]$, with 0 and 1 being the worst and best possible, respectively, empirical qualities of hyperparameters.

5.3 Experimental Results

We evaluated the hyperparameter sensitivity of SiblingRivalry by running the autotuner on a set of four benchmarks: Sort, Bin Packing, Image Compression and Poisson, with twenty different combinations of C and W values each. For each run, we measured the metrics described in Section 5.2. We performed the tests on the `Xeon8` and `AMD48` systems (see Table 4.1). The reported numbers for `Xeon8` have been averaged over 30 runs, and the numbers for `AMD48` over 20 runs. All experiments were performed in a "cold start" scenario, where the initial configuration was *not* tuned offline before invoking SiblingRivalry.

All results for a given metric have been normalized with respect to the best mean value of that metric. If the metric is positive, i.e. if higher values are better (e.g.

	Static System				Dynamic System			
	Xeon8		AMD48		Xeon8		AMD48	
	C	W	C	W	C	W	C	W
Sort	50.00	5	5.00	5	5.00	5	5.00	5
Bin Packing	0.01	5	0.10	5	5.00	500	5.00	500
Poisson	50.00	500	50.00	500	0.01	500	5.00	5
Image Compression	0.10	100	50.00	50	0.01	100	50.00	50

(a) Best performing values of the hyperparameters C and W over an empirical sample.

	Static System		Dynamic System	
	Xeon8	AMD48	Xeon8	AMD48
Sort	0.8921	0.8453	0.9039	0.9173
Bin Packing	0.8368	0.8470	0.9002	0.9137
Poisson	0.8002	0.8039	0.8792	0.6285
Image Compression	0.9538	0.9897	0.9403	0.9778

(b) Scores of the best performing hyperparameters.

Figure 5-1: Best performing hyperparameters and associated score function values under the Static System and Dynamic System autotuning scenarios on `Xeon8` and `AMD48` architectures.

throughput), then the best configuration will have a mean of 1 and the suboptimal configurations will be in the range $[0; 1)$. If the metric is negative, i.e. if lower values are better (e.g. RMSE), the best configuration will have a mean of 1 and the suboptimal configurations will be in the range $(1; +\infty)$. The constant by which the metrics have been divided for normalization is reported as the "normalization factor". We additionally performed a significance test, for each metric, to find configurations which were not significantly different from the best (p-value ≥ 0.05). Such configurations are marked with an asterisk $*$. We used a Wilcoxon signed-rank non-parametric test for this purpose, since the majority of our data was not normally distributed (see Sections A.1 and A.2 for details).

In Sections 5.3.1, 5.3.2, 5.3.3 and 5.3.4 we evaluate the four benchmarks along individual metrics, with Figure 5-1 listing optimal hyperparameter values under the score functions of the Static and Dynamic System scenarios. Finally, in Section 5.4 we discuss the feasibility and attempt to find globally optimal hyperparameters that work reasonably well for all the benchmarks.

5.3.1 Sort

Figure 5-2 shows the performance of the Sort benchmark on the Xeon8 system. Mean throughput is maximized for the configuration $(C, W) = (50, 100)$ (Table 5-2(a)). It appears throughput increases with the value of C, with the value of W having little effect. For example, the configuration $(C, W) = (50, 5)$ performs within 4% of the optimal $(C, W) = (50, 100)$ despite having a W an order of magnitude smaller. Best candidate throughput looks similar (Table 5-2(b), with best candidate found at $(C, W) = (50, 500)$. The strong dependence on C and lack of dependence on W is also evident, and suggests that Sort benefits from a highly exploitative behavior over a short window size. This dependency trend is reversed in the convergence times (Table 5-2(c)). Smaller values of W result in faster convergence with an optimum at $(C, W) = (5, 5)$.

The trends on the AMD48 system appear similar. While the mean and best candidate throughputs have different optima (both at $(C, W) = (50, 50)$, see Table 5-3(a)), had we used Xeon8's optima instead we would get a performance degradation of only 1%. The convergence time optimum is the same on both architectures, $(C, W) = (5, 5)$ (Tables 5-2(c) 5-3(c)). The optima with respect to the Static and Dynamic System scores are shown in Figure 5-1.

5.3.2 Bin Packing

The behavior of the Bin Packing benchmark on both the Xeon8 and AMD48 system is similar, especially with respect to suboptimal hyperparameter configurations (Figures 5-4 and 5-5).

On the Xeon8, average throughput is maximized for $(C, W) = (0.5, 50)$ (Table 5-4(a)) while on the AMD48 the optimum is at $(C, W) = (5, 500)$ (Table 5-5(a)). However, we could have confidently used any of those configurations on both systems and gotten a mean throughput within 4% of optimum. Other throughput patterns were evident as well. For example, on both systems configurations with a large C and/or small W had the lowest mean throughput.

Surprisingly, configurations with a very low mean throughput were the ones for which the fastest candidates were found (Tables 5-4(b) and 5-5(b)). For example, while $(C, W) = (0.1, 5)$ found the fastest candidate on the AMD48 system, and a candidate within 8% of fastest on the Xeon8, the average throughputs were below 10% of optimum on both systems. This is due to the relatively slow convergence of those configurations (Tables 5-4(e) and 5-5(e)). In fact, all of the configurations with the best candidate within 20% of optimum, corresponding to $W = 5$ on both the Xeon8 and AMD48, had convergence times at least twice as slow as optimum, with some up to 7x slower.

Configurations with the best candidates performed worst with respect to meeting their accuracy target, which is to be expected due to long convergence. However, the RMSE (Tables 5-4(c) and 5-5(c)) was relatively low for those configurations, which suggests that while they were not meeting the target exactly, they managed to stay close to it for most of the run. This is in contrast to configuration with a high average throughput, which demonstrated very large values of RMSE.

The optima with respect to the Static and Dynamic System scores are shown in Figure 5-1.

5.3.3 Poisson

Figure 5-6 shows the performance of the Poisson benchmark on the Xeon8 system. Mean throughput is maximized for the configuration $(C, W) = (5, 50)$, although due to the large variance many other configurations could be optimal (Table 5-6(a)). Configurations with a high value of W, while slower than optimum, demonstrate a relatively consistent throughput across runs. $(C, W) = (50, 500)$, for example, is within 35% of optimum with a standard deviation of only 20%. This is also the configuration for which the best candidate was found (Table 5-6(b)). The quality of the best candidate seems to increase with the value of C and W, and suggests that Poisson benefits from a highly exploratory behavior, with occasional exploitation over a long mutator window.

The accuracy error for the Poisson benchmark on the Xeon8 remains large but

relatively stable across all the tested hyperparameter configurations, with an optimum at $(C, W) = (50, 500)$ (Table 5-6(c)). Likewise, over 99% of generations met the accuracy target, regardless of C and W (Table 5-6(d)).

`Xeon8` convergence times displayed a significantly greater variation across hyperparameter values (Table 5-6(e). Smaller values of C and W seemed to perform much better overall (up to a factor of over 900x), with an optimum at $(C, W) = (5, 100)$. However, their faster convergence was at a noticeable expense of throughput and best candidate's quality: the optimal configuration in terms of convergence had over 40% slower overall throughput and best candidate that was over 50% slower. This behavior suggests that estimating proper mutator weights for the Poisson benchmark on the `Xeon8` is a difficult task which takes a very long time, but which eventually results in better quality candidates.

Performance on the `AMD48` system (Figure 5-7) was similar in terms of accuracy, with over 97% of all candidates meeting the accuracy target across different hyperparameter values (Table 5-7(d). The RMSE error was slightly larger compared to the `Xeon8` system (Table 5-7(c)). In term of throughput, while the `AMD48` had optimal average throughput at $(C, W) = (5, 500)$ compared to `Xeon8`'s $(C, W) = (5, 50)$ (5-7(a) and Table 5-6(a)), a significant variance of measurements on both systems could potentially account for the difference.

The autotuner on the `AMD48` system found a candidate with the best throughput at $(C, W) = (50, 500)$ (Table 5-7(b)), which is the same as on the `Xeon8` system. `AMD48`'s convergence times displayed a significant variation, with the fastest convergence at $(C, W) = (5, 5)$, and $\approx$51x slower at $(C, W) = (50, 500)$ (Table 5-7(e)). However, the fast convergence of the former configuration meant that worse overall candidates were found, with mean throughput only at about 25% of optimum. Note that compared to the `Xeon8` there are significantly more slow-converging configurations.

The optima with respect to the score functions are shown in Figure 5-1.

5.3.4 Image Compression

Image Compressionis a relatively easy benchmark to autotune, and the data for both the `Xeon8` (Figure 5-8) and the `AMD48` systems (Figure 5-9) reflects that: all configurations perform well regardless of hyperparameter values. The only (slight) exception is the convergence time on the `Xeon8` system, where the slowest configuration $(C, W) = (5, 5)$ takes about 85% longer to converge than the optimal $(C, W) = (0.01, 100)$ (Table 5-8(e)). When one looks at absolute instead of relative values, however, this amounts to only abound 2-3 races, which in most use cases would not be significant.

Based on the data, the user could confidently choose most of the configuration we evaluated in both the static and dynamic system scenarios and get good average performance on all the metrics. The optima with respect to the score functions are shown in Figure 5-1.

5.4 The Big Picture

Ideally, the user would be able to repeat our sampling experiments for different hyperparameter configurations, and select values of C and W based on their actual performance on the given program. This requirement is, however, unrealistic. Accurately evaluating a single program by testing many configurations averaged over a considerable number of runs could take hours, if not days. This high upfront cost could make the use of SiblingRivalry infeasible in many settings.

There are many solutions to this problem. First, SiblingRivalry could automatically derive a fast model of the tuned program, and optimize hyperparameters based on that model. While intriguing, building such a model would require substantial changes to SiblingRivalry, and might be highly suboptimal due to inaccuracies of the model. A second solution, which we adopt, is to find "globally optimal" values of hyperparameters, i.e. hyperparameters with a track record of performing well across benchmarks. One might think this feat impossible, since the optima for different benchmarks differ significantly (see Figure 5-1). However, as we will soon demonstrate, by sacrificing only a little performance on each benchmark, we can indeed find

globally optimal hyperparameters.

5.4.1 Globally Optimal Hyperparameters

We used the score functions from the Static and Dynamic scenarios to find hyperparameters that maximized the mean score on all the benchmarks. We found that the hyperparameters $(C, W) = (5, 5)$ for the Static System and $(C, W) = (5, 100)$ for the Dynamic System maximized this score. The results are shown in Table 5.1. For the sake of illustration, we normalized each score with respect to the optimum for the given benchmark and scenario (Table 5-1(b)). Despite fixing hyperparameter values across benchmarks, we got a mean score of 0.8832 for the Static and 0.8245 for the Dynamic System, which means that we only sacrificed less than 20% of the performance by not tuning hyperparameters on a per-benchmark basis. This result implies that the hyperparameters we found are likely to generalize to other benchmarks, thus providing sensible defaults and removing the need to optimize them on a per-program basis.

	Static System		**Dynamic System**	
	Xeon8	**AMD48**	**Xeon8**	**AMD48**
Sort	0.9571	1.0000	0.7416	0.6112
Bin Packing	0.8561	0.9472	0.6742	0.8874
Poisson	0.7064	0.7109	0.9077	0.9607
Image Compression	0.9244	0.9635	0.8992	0.9142

Table 5.1: Benchmark scores for the globally optimal values of hyperparameters normalized with respect to the best score for the given benchmark and scenario. The hyperparameters were $C = 5$, $W = 5$ for the Static System, and $C = 5$, $W = 100$ for the Dynamic System. Mean scores are 0.8832 and 0.8245 for the Static and Dynamic systems, respectively.

5.5 Conclusions

We performed a detailed experimental investigation of hyperparameter values on SiblingRivalry's autotuning performance. We evaluated four benchmarks with respect

to five metrics, and demonstrated that optimal hyperparameter values differ significantly between benchmarks. We also showed how two possible autotuning scenarios can affect the optimal hyperparameter values. We further demonstrated that a single choice of hyperparameters across many benchmarks is possible, with only a small performance degradation. Such a choice provides sensible defaults for SiblingRivalry, removing the need for the user to tune them per-program, and thus making our approach more feasible in a real-world setting.

C \ W	5	50	100	500
0.01	0.7022 ±0.1625	0.7892 ±0.1459	0.7411 ±0.1667	0.7682 ±0.1532
0.10	0.7512 ±0.1320	0.7538 ±0.1558	0.7829 ±0.1249	0.7680 ±0.1677
0.50	0.7398 ±0.1776	0.7958 ±0.1278	0.7956 ±0.1605	0.7579 ±0.1521
5.00	0.8077 ±0.1609	0.9319 ±0.0932	0.9521 ±0.0779	0.9562 ±0.0653
50.00	0.9671 ±0.0371	∗ 0.9976 ±0.0104	**1.0000** ±0.0076	∗ 0.9961 ±0.0154

(a) Mean throughput (races/s) normalized to the fastest configuration. Normalization factor: 5.4741.

C \ W	5	50	100	500
0.01	0.7171 ±0.1467	0.7981 ±0.1318	0.7594 ±0.1469	0.7870 ±0.1416
0.10	0.7627 ±0.1077	0.7624 ±0.1458	0.7917 ±0.1174	0.7914 ±0.1506
0.50	0.7577 ±0.1506	0.7997 ±0.1149	0.8023 ±0.1494	0.7867 ±0.1341
5.00	0.8174 ±0.1347	0.9295 ±0.0882	0.9479 ±0.0706	0.9619 ±0.0652
50.00	0.9602 ±0.0334	0.9968 ±0.0057	∗ 0.9981 ±0.0034	**1.0000** ±0.0041

(b) Best candidate throughput (races/s) normalized to the fastest candidate. Normalization factor: 6.2435.

C \ W	5	50	100	500
0.01	∗ 1.4980 ±1.4224	2.3441 ±2.8144	2.9156 ±2.1436	6.9268 ±6.9161
0.10	2.2189 ±3.1541	2.5787 ±2.3936	3.5260 ±2.1447	4.1217 ±3.8916
0.50	∗ 1.6790 ±2.4116	1.6802 ±1.2014	2.2744 ±1.5285	6.7082 ±5.2096
5.00	**1.0000** ±1.0748	2.0503 ±1.1397	2.5746 ±1.8791	6.6426 ±5.6713
50.00	1.6134 ±0.8930	3.7238 ±2.5726	3.0304 ±1.7031	4.9711 ±5.2824

(c) Time to convergence (number of races) normalized to the lowest one achieved. Normalization factor: 115.2667.

Figure 5-2: Metrics for benchmark `Sort` on the `Xeon8` system evaluated with different values of hyperparameters. An asterisk $*$ next to a number means that the difference from optimum is not statistically significant (p-value ≥ 0.05).

C \ W	5	50	100	500
0.01	0.7634 ±0.1949	0.8859 ±0.0900	0.8881 ±0.1452	0.8478 ±0.0841
0.10	0.7759 ±0.1291	0.8034 ±0.1279	0.7601 ±0.1965	0.8827 ±0.0795
0.50	0.7576 ±0.1890	0.7951 ±0.1692	0.8482 ±0.0935	0.8683 ±0.0941
5.00	0.8346 ±0.0923	0.9426 ±0.0685	∗ 0.9841 ±0.0536	∗ 0.9831 ±0.0291
50.00	0.9732 ±0.0143	**1.0000** ±0.0154	∗ 0.9926 ±0.0132	0.9798 ±0.0290

(a) Mean throughput (races/s) normalized to the fastest configuration. Normalization factor: 3.4538.

C \ W	5	50	100	500
0.01	0.7631 ±0.1738	0.8648 ±0.0947	0.8723 ±0.1451	0.8367 ±0.1007
0.10	0.7596 ±0.1172	0.7899 ±0.1227	0.7484 ±0.1833	0.8746 ±0.0894
0.50	0.7625 ±0.1704	0.7851 ±0.1563	0.8415 ±0.0745	0.8541 ±0.0988
5.00	0.8067 ±0.0948	0.9284 ±0.0773	∗ 0.9750 ±0.0573	0.9698 ±0.0352
50.00	0.9600 ±0.0231	**1.0000** ±0.0200	∗ 0.9950 ±0.0170	0.9838 ±0.0120

(b) Best candidate throughput (races/s) normalized to the fastest candidate. Normalization factor: 3.8713.

C \ W	5	50	100	500
0.01	∗ 4.9463 ±9.3636	2.4251 ±3.8180	6.0850 ±8.0789	∗ 4.5145 ±8.4378
0.10	∗ 3.4564 ±7.5355	∗ 2.8098 ±6.5613	∗ 1.7271 ±1.9581	8.4430 ±13.9749
0.50	∗ 5.6018 ±9.9619	3.6242 ±5.3498	∗ 4.4474 ±8.8197	4.5727 ±7.4949
5.00	**1.0000** ±0.8090	5.7069 ±10.2162	7.2841 ±8.9958	7.3154 ±8.7998
50.00	4.8658 ±2.5240	16.0716 ±16.1863	12.9374 ±10.9121	6.9687 ±5.3936

(c) Time to convergence (number of races) normalized to the lowest one achieved. Normalization factor: 22.3500.

Figure 5-3: Metrics for benchmark Sort on the AMD48 system evaluated with different values of hyperparameters. An asterisk $*$ next to a number means that the difference from optimum is not statistically significant (p-value ≥ 0.05).

C \ W	5	50	100	500
0.01	0.0467 ±0.0173	∗ 0.5949 ±0.3913	∗ 0.5839 ±0.4003	∗ 0.6476 ±0.3909
0.10	0.0509 ±0.0198	∗ 0.7642 ±0.9978	∗ 0.5285 ±0.3681	∗ 0.6368 ±0.3433
0.50	0.0451 ±0.0158	**1.0000** ±1.8869	∗ 0.6154 ±0.3043	∗ 0.8782 ±1.0515
5.00	0.0445 ±0.0182	∗ 0.5796 ±0.5914	∗ 0.9183 ±1.1676	0.9796 ±1.0238
50.00	0.1153 ±0.0511	0.2661 ±0.1583	0.3333 ±0.2218	0.2504 ±0.0695

(a) Mean throughput (races/s) normalized to the fastest configuration. Normalization factor: 48.0584.

C \ W	5	50	100	500
0.01	**1.0000** ±0.0295	0.6268 ±0.2227	0.5813 ±0.2001	0.4972 ±0.0140
0.10	∗ 0.9285 ±0.1854	0.5983 ±0.2154	0.5439 ±0.1461	0.4963 ±0.0139
0.50	∗ 0.9062 ±0.2038	0.5598 ±0.1717	0.4938 ±0.0134	0.4969 ±0.0133
5.00	∗ 0.8120 ±0.2546	0.6827 ±0.2451	0.6235 ±0.2142	0.5114 ±0.0088
50.00	∗ 0.8444 ±0.2495	0.5327 ±0.0940	0.5162 ±0.0027	0.5163 ±0.0029

(b) Best candidate throughput (races/s) normalized to the fastest candidate. Normalization factor: 3.8151.

C \ W	5	50	100	500
0.01	**1.0000** ±0.1467	2.7451 ±0.9020	2.6741 ±0.9574	2.9507 ±0.8128
0.10	∗ 1.0548 ±0.1710	2.8627 ±0.9703	2.7011 ±0.8725	3.0573 ±0.8367
0.50	∗ 1.0156 ±0.1620	2.7960 ±1.1384	3.0342 ±1.0395	2.9594 ±1.0789
5.00	∗ 1.0190 ±0.1604	3.0010 ±1.2843	3.0041 ±1.2071	3.6437 ±1.2162
50.00	1.5095 ±0.2146	2.1523 ±0.3023	2.4795 ±0.5332	2.1054 ±0.1756

(c) Accuracy Root Mean Square Error (RMSE) normalized to the lowest one achieved. Normalization factor: 0.0408.

C \ W	5	50	100	500
0.01	0.3207 ±0.5332	0.8553 ±0.2299	0.8068 ±0.2435	0.7804 ±0.1866
0.10	0.5514 ±0.5944	0.8192 ±0.2390	0.8461 ±0.2151	0.7987 ±0.1907
0.50	0.3636 ±0.5170	0.7839 ±0.3415	0.6935 ±0.2701	0.6685 ±0.3585
5.00	0.3719 ±0.5280	0.7332 ±0.3096	0.7156 ±0.3169	0.5623 ±0.2938
50.00	**1.0000** ±0.2889	∗ 0.9681 ±0.1159	0.9380 ±0.1239	0.9804 ±0.1273

(d) Percentage of races that met accuracy target normalized to the highest one achieved. Normalization factor: 5.7006.

C \ W	5	50	100	500
0.01	5.4381 ±3.1205	∗ 3.5652 ±5.0255	∗ 1.6026 ±2.7460	∗ 1.3418 ±2.0867
0.10	3.6426 ±2.8052	2.9784 ±4.0732	∗ 1.9122 ±3.0710	∗ 1.2129 ±1.7783
0.50	6.2328 ±4.2491	∗ 10.8195 ±49.485	**1.0000** ±0.6316	∗ 5.7735 ±16.4545
5.00	2.9953 ±2.5389	4.4862 ±5.8300	3.3855 ±4.8506	∗ 1.2183 ±1.3415
50.00	7.0258 ±4.8670	∗ 1.4899 ±1.7110	∗ 1.0042 ±0.7444	∗ 1.2749 ±0.9714

(e) Time to convergence (number of races) normalized to the lowest one achieved. Normalization factor: 74.7500.

Figure 5-4: Metrics for benchmark Bin Packing on the Xeon8 system evaluated with different values of hyperparameters. An asterisk ∗ next to a number means that the difference from optimum is not statistically significant (p-value ≥ 0.05).

C \ W	5	50	100	500
0.01	0.0831 ±0.0364	0.6379 ±0.4622	∗ 0.8305 ±0.3674	∗ 0.7695 ±0.5082
0.10	0.0844 ±0.0330	∗ 0.8128 ±0.3908	∗ 0.9066 ±0.9946	∗ 0.8326 ±0.5185
0.50	0.0803 ±0.0354	∗ 0.9794 ±1.1818	∗ 0.7285 ±0.4457	∗ 0.7268 ±0.4810
5.00	0.1057 ±0.0395	∗ 0.8514 ±0.7817	∗ 0.7541 ±0.4020	**1.0000** ±0.6079
50.00	0.2465 ±0.1100	0.3234 ±0.1043	0.3109 ±0.0782	0.3255 ±0.0692

(a) Mean throughput (races/s) normalized to the fastest configuration. Normalization factor: 13.9339.

C \ W	5	50	100	500
0.01	∗ 0.8448 ±0.2431	0.6802 ±0.2673	0.4688 ±0.1304	0.4404 ±0.0125
0.10	**1.0000** ±0.0186	0.5427 ±0.2076	0.4723 ±0.1164	0.4378 ±0.0137
0.50	∗ 0.8996 ±0.2226	∗ 0.5856 ±0.2359	0.4934 ±0.1624	0.4351 ±0.0105
5.00	∗ 0.9479 ±0.1480	∗ 0.5683 ±0.2190	0.4564 ±0.0071	0.4575 ±0.0080
50.00	∗ 0.6503 ±0.2625	0.4606 ±0.0016	0.4595 ±0.0035	0.4590 ±0.0035

(b) Best candidate throughput (races/s) normalized to the fastest candidate. Normalization factor: 2.0105.

C \ W	5	50	100	500
0.01	∗ 1.0209 ±0.1538	2.3683 ±0.8644	3.0144 ±0.5231	2.7324 ±0.8927
0.10	∗ 1.0083 ±0.1280	2.8437 ±0.7947	2.6830 ±1.0425	2.7739 ±0.9000
0.50	**1.0000** ±0.1384	2.3480 ±1.0114	2.5601 ±0.8720	2.5438 ±0.8733
5.00	∗ 1.1011 ±0.1719	2.0680 ±0.9162	3.0481 ±1.0211	2.7223 ±0.7398
50.00	1.6999 ±0.3230	1.9277 ±0.1941	1.9621 ±0.1563	1.9383 ±0.1063

(c) Accuracy Root Mean Square Error (RMSE) normalized to the lowest one achieved. Normalization factor: 0.0431.

C \ W	5	50	100	500
0.01	0.3984 ±0.5581	0.8056 ±0.2371	0.6592 ±0.1896	0.7184 ±0.2518
0.10	0.2532 ±0.4398	0.7751 ±0.2169	0.7691 ±0.3175	0.6256 ±0.2530
0.50	0.2736 ±0.4794	0.7882 ±0.3089	0.7443 ±0.3214	0.7162 ±0.2481
5.00	∗ 0.6646 ±0.5495	0.6791 ±0.3632	0.6800 ±0.2176	0.7191 ±0.2569
50.00	∗ 0.9611 ±0.1635	∗ 0.9066 ±0.2240	∗ 0.9639 ±0.2143	**1.0000** ±0.1567

(d) Percentage of races that met accuracy target normalized to the highest one achieved. Normalization factor: 5.9188.

C \ W	5	50	100	500
0.01	∗ 2.7844 ±1.9587	3.5227 ±3.2427	∗ 1.4105 ±1.3325	∗ 1.7656 ±2.9000
0.10	4.2545 ±2.3638	∗ 2.4973 ±3.7407	∗ 1.2115 ±1.6913	1.8967 ±2.1635
0.50	3.3455 ±1.4152	∗ 3.9990 ±9.0836	∗ 1.8344 ±1.9539	**1.0000** ±0.6515
5.00	4.5455 ±2.2751	∗ 2.2498 ±2.7134	∗ 1.1527 ±0.6035	∗ 1.2086 ±0.6339
50.00	2.6691 ±2.2855	∗ 1.4236 ±1.3947	∗ 1.1727 ±0.9371	∗ 1.1400 ±0.4892

(e) Time to convergence (number of races) normalized to the lowest one achieved. Normalization factor: 55.0000.

Figure 5-5: Metrics for benchmark Bin Packing on the AMD48 system evaluated with different values of hyperparameters. An asterisk ∗ next to a number means that the difference from optimum is not statistically significant (p-value ≥ 0.05).

C \ W	5	50	100	500
0.01	0.4411 ±0.0083	* 0.4941 ±0.0703	* 0.5321 ±0.1314	0.9875 ±0.7413
0.10	* 0.4419 ±0.0077	* 0.7453 ±1.2946	* 0.5312 ±0.1390	* 0.7656 ±0.5697
0.50	* 0.4419 ±0.0059	* 0.4663 ±0.0529	* 0.4746 ±0.0819	* 0.6493 ±0.4313
5.00	0.4416 ±0.0051	**1.0000** ±2.0439	* 0.5961 ±0.7323	* 0.5783 ±0.4994
50.00	* 0.5059 ±0.1776	0.5312 ±0.1409	0.8636 ±0.7448	0.6529 ±0.1996

(a) Mean throughput (races/s) normalized to the fastest configuration. Normalization factor: 11.5018.

C \ W	5	50	100	500
0.01	0.4753 ±0.0005	0.4756 ±0.0006	0.4757 ±0.0006	0.4761 ±0.0007
0.10	0.4754 ±0.0005	0.4755 ±0.0007	0.5035 ±0.1497	0.5376 ±0.2456
0.50	0.4753 ±0.0005	0.4754 ±0.0005	0.4755 ±0.0005	0.5697 ±0.2898
5.00	0.4753 ±0.0004	0.4755 ±0.0006	0.4755 ±0.0006	0.5254 ±0.2672
50.00	0.4774 ±0.0101	0.5602 ±0.2333	0.6241 ±0.3894	**1.0000** ±0.5139

(b) Best candidate throughput (races/s) normalized to the fastest candidate. Normalization factor: 11.6923.

C \ W	5	50	100	500
0.01	1.2947 ±0.0000	1.2939 ±0.0013	1.2939 ±0.0012	1.2884 ±0.0081
0.10	1.2947 ±0.0001	1.2920 ±0.0118	1.2646 ±0.1587	1.2420 ±0.1848
0.50	1.2947 ±0.0000	1.2945 ±0.0005	1.2942 ±0.0018	1.2286 ±0.1929
5.00	1.2947 ±0.0001	1.2896 ±0.0185	1.2921 ±0.0127	1.2592 ±0.1803
50.00	1.2937 ±0.0021	1.2563 ±0.1785	1.1897 ±0.2504	**1.0000** ±0.3300

(c) Accuracy Root Mean Square Error (RMSE) normalized to the lowest one achieved. Normalization factor: 18.5365.

C \ W	5	50	100	500
0.01	* 1.0000 ±0.0001	0.9985 ±0.0021	0.9986 ±0.0020	0.9897 ±0.0131
0.10	* 1.0000 ±0.0001	0.9956 ±0.0189	0.9988 ±0.0017	0.9946 ±0.0097
0.50	**1.0000** ±0.0001	0.9996 ±0.0008	0.9991 ±0.0029	0.9967 ±0.0080
5.00	* 1.0000 ±0.0001	0.9918 ±0.0295	0.9958 ±0.0204	0.9967 ±0.0134
50.00	0.9986 ±0.0031	0.9968 ±0.0045	0.9836 ±0.0261	0.9982 ±0.0023

(d) Percentage of races that met accuracy target normalized to the highest one achieved. Normalization factor: 99.9989.

C \ W	5	50	100	500
0.01	1.2432 ±0.4555	1.1622 ±0.4531	1.1081 ±0.3907	1.2973 ±0.7130
0.10	1.1622 ±0.4991	1.2703 ±0.6843	25.4324 ±130.94	89.1351 ±378.01
0.50	1.0811 ±0.4358	1.1081 ±0.4433	1.1892 ±0.5013	144.2973 ±446.11
5.00	1.0811 ±0.3822	1.4324 ±0.8291	**1.0000** ±0.3429	195.2162 ±1044.50
50.00	8.8378 ±40.5218	158.6486 ±522.98	174.4324 ±506.73	967.9730 ±1087.86

(e) Time to convergence (number of races) normalized to the lowest one achieved. Normalization factor: 1.2333.

Figure 5-6: Metrics for benchmark Poisson on the Xeon8 system evaluated with different values of hyperparameters. An asterisk $*$ next to a number means that the difference from optimum is not statistically significant (p-value ≥ 0.05).

C \ W	5	50	100	500
0.01	0.2558 ±0.0057	* 0.3311 ±0.1063	* 0.6048 ±0.9884	* 0.7500 ±0.6350
0.10	0.2548 ±0.0029	* 0.5237 ±0.9201	* 0.3042 ±0.0883	* 0.6913 ±0.7247
0.50	0.2562 ±0.0048	0.2798 ±0.0771	* 0.3175 ±0.1334	0.3584 ±0.3642
5.00	0.2570 ±0.0072	* 0.7374 ±1.4427	0.2794 ±0.0644	**1.0000** ±1.4595
50.00	0.2717 ±0.0172	* 0.4783 ±0.2569	* 0.4395 ±0.2083	* 0.3819 ±0.1225

(a) Mean throughput (races/s) normalized to the fastest configuration. Normalization factor: 11.5793.

C \ W	5	50	100	500
0.01	0.4705 ±0.0074	0.5314 ±0.2673	0.4685 ±0.0081	* 0.5422 ±0.3107
0.10	0.4664 ±0.0080	0.4670 ±0.0078	* 0.4698 ±0.0061	0.6097 ±0.4197
0.50	0.4655 ±0.0085	0.4683 ±0.0075	0.5001 ±0.1500	0.4688 ±0.0059
5.00	0.4644 ±0.0076	0.4688 ±0.0070	0.4656 ±0.0074	0.4762 ±0.0215
50.00	0.4661 ±0.0085	* 0.6705 ±0.4763	0.4683 ±0.0073	**1.0000** ±0.6763

(b) Best candidate throughput (races/s) normalized to the fastest candidate. Normalization factor: 6.8888.

C \ W	5	50	100	500
0.01	1.1459 ±0.0001	1.1078 ±0.1597	1.1366 ±0.0270	* 1.1115 ±0.0944
0.10	1.1460 ±0.0001	1.1390 ±0.0238	1.1447 ±0.0023	1.1052 ±0.1035
0.50	1.1459 ±0.0001	1.1453 ±0.0020	1.1067 ±0.1663	1.1431 ±0.0091
5.00	1.1459 ±0.0002	1.1346 ±0.0335	1.1435 ±0.0095	1.1194 ±0.0426
50.00	1.1455 ±0.0004	* 1.0301 ±0.2605	* 1.1367 ±0.0126	**1.0000** ±0.2382

(c) Accuracy Root Mean Square Error (RMSE) normalized to the lowest one achieved. Normalization factor: 20.9425.

C \ W	5	50	100	500
0.01	* 0.9999 ±0.0002	0.9973 ±0.0036	0.9833 ±0.0478	0.9768 ±0.0305
0.10	**1.0000** ±0.0001	0.9874 ±0.0422	0.9976 ±0.0042	0.9803 ±0.0349
0.50	* 0.9999 ±0.0002	* 0.9988 ±0.0037	0.9979 ±0.0047	* 0.9948 ±0.0166
5.00	* 0.9999 ±0.0003	0.9799 ±0.0593	0.9955 ±0.0171	0.9610 ±0.0712
50.00	0.9991 ±0.0008	0.9896 ±0.0136	0.9831 ±0.0230	0.9965 ±0.0032

(d) Percentage of races that met accuracy target normalized to the highest one achieved. Normalization factor: 99.9971.

C \ W	5	50	100	500
0.01	4.8419 ±7.0167	4.3787 ±8.7464	5.7390 ±13.6597	7.0000 ±16.1657
0.10	* 2.4154 ±5.4030	* 2.3676 ±5.6973	* 5.1875 ±11.9706	14.3309 ±31.7173
0.50	* 1.7794 ±3.5586	* 5.1838 ±10.1641	* 1.8272 ±5.5271	* 3.5919 ±5.7688
5.00	**1.0000** ±1.8098	3.7243 ±5.5670	* 1.0772 ±2.1406	6.0441 ±12.0280
50.00	* 4.6176 ±11.1570	23.2390 ±50.5314	6.7941 ±12.5596	50.9118 ±60.2539

(e) Time to convergence (number of races) normalized to the lowest one achieved. Normalization factor: 13.6000.

Figure 5-7: Metrics for benchmark Poisson on the AMD48 system evaluated with different values of hyperparameters. An asterisk $*$ next to a number means that the difference from optimum is not statistically significant (p-value ≥ 0.05).

C \ W	5	50	100	500
0.01	∗ 0.9341 ±0.2195	∗ 0.9657 ±0.1600	∗ 0.8805 ±0.3002	**1.0000** ±0.0158
0.10	∗ 0.7971 ±0.3753	0.8172 ±0.3513	∗ 0.9929 ±0.0290	∗ 0.9347 ±0.2201
0.50	∗ 0.9998 ±0.0143	∗ 0.9679 ±0.1584	∗ 0.9647 ±0.1591	0.8469 ±0.3268
5.00	∗ 0.9733 ±0.1604	0.9052 ±0.2626	0.8500 ±0.3287	0.8771 ±0.2984
50.00	0.9626 ±0.1578	∗ 0.9628 ±0.1573	0.9347 ±0.2192	0.9319 ±0.2190

(a) Mean throughput (races/s) normalized to the fastest configuration. Normalization factor: 0.1514.

C \ W	5	50	100	500
0.01	∗ 0.9363 ±0.2217	∗ 0.9646 ±0.1596	∗ 0.8789 ±0.3021	∗ 0.9995 ±0.0114
0.10	∗ 0.7923 ±0.3770	0.8166 ±0.3539	∗ 0.9969 ±0.0143	0.9330 ±0.2205
0.50	**1.0000** ±0.0099	0.9627 ±0.1588	∗ 0.9678 ±0.1599	0.8461 ±0.3296
5.00	∗ 0.9668 ±0.1596	0.9033 ±0.2649	∗ 0.8508 ±0.3316	∗ 0.8787 ±0.3020
50.00	0.9628 ±0.1594	∗ 0.9651 ±0.1592	∗ 0.9381 ±0.2217	∗ 0.9361 ±0.2216

(b) Best candidate throughput (races/s) normalized to the fastest candidate. Normalization factor: 0.1732.

C \ W	5	50	100	500
0.01	∗ 1.0023 ±0.0101	∗ 1.0015 ±0.0066	1.0017 ±0.0126	∗ 1.0010 ±0.0053
0.10	∗ 1.0001 ±0.0038	∗ 1.0031 ±0.0106	∗ 1.0039 ±0.0124	∗ 1.0029 ±0.0097
0.50	∗ 1.0007 ±0.0038	∗ 1.0010 ±0.0045	∗ 1.0021 ±0.0095	∗ 1.0023 ±0.0095
5.00	∗ 1.0004 ±0.0042	1.0021 ±0.0050	**1.0000** ±0.0034	∗ 1.0010 ±0.0040
50.00	∗ 1.0024 ±0.0058	∗ 1.0017 ±0.0046	∗ 1.0017 ±0.0052	1.0042 ±0.0074

(c) Accuracy Root Mean Square Error (RMSE) normalized to the lowest one achieved. Normalization factor: 0.1017.

C \ W	5	50	100	500
0.01	**1.0000** ±0.0000	**1.0000** ±0.0000	**1.0000** ±0.0000	**1.0000** ±0.0000
0.10	**1.0000** ±0.0000	**1.0000** ±0.0000	**1.0000** ±0.0000	**1.0000** ±0.0000
0.50	**1.0000** ±0.0000	**1.0000** ±0.0000	**1.0000** ±0.0000	**1.0000** ±0.0000
5.00	**1.0000** ±0.0000	**1.0000** ±0.0000	**1.0000** ±0.0000	**1.0000** ±0.0000
50.00	**1.0000** ±0.0000	**1.0000** ±0.0000	**1.0000** ±0.0000	**1.0000** ±0.0000

(d) Percentage of races that met accuracy target normalized to the highest one achieved. Normalization factor: 100.0000.

C \ W	5	50	100	500
0.01	1.2432 ±0.8066	1.3333 ±0.9413	**1.0000** ±0.4942	1.3604 ±0.8032
0.10	1.1802 ±0.6490	1.0270 ±0.7496	1.2793 ±0.7080	∗ 1.1982 ±0.8277
0.50	1.6396 ±2.1073	1.3243 ±0.7264	1.5045 ±0.9959	∗ 1.2162 ±0.8792
5.00	1.8468 ±1.4217	∗ 1.2162 ±0.7311	1.1892 ±0.6419	1.4775 ±0.9961
50.00	1.4865 ±1.0297	1.3153 ±0.7540	1.6036 ±1.5304	1.7027 ±3.0194

(e) Time to convergence (number of races) normalized to the lowest one achieved. Normalization factor: 3.7000.

Figure 5-8: Metrics for benchmark Image Compression on the Xeon8 system evaluated with different values of hyperparameters. An asterisk ∗ next to a number means that the difference from optimum is not statistically significant (p-value ≥ 0.05).

C \ W	5	50	100	500
0.01	∗ 0.9566 ±0.1929	∗ 0.9960 ±0.0123	∗ 0.9540 ±0.1929	∗ 0.9521 ±0.1920
0.10	∗ 0.9923 ±0.0264	∗ 0.9377 ±0.1928	∗ 0.9989 ±0.0125	∗ 0.8600 ±0.3135
0.50	∗ 0.9940 ±0.0378	∗ 0.9929 ±0.0262	∗ 0.9991 ±0.0103	∗ 0.9488 ±0.1920
5.00	∗ 0.9938 ±0.0276	0.9522 ±0.1916	∗ 0.9102 ±0.2650	∗ 0.9101 ±0.2642
50.00	∗ 0.9803 ±0.0480	**1.0000** ±0.0097	0.9054 ±0.2630	∗ 0.9496 ±0.1918

(a) Mean throughput (races/s) normalized to the fastest configuration. Normalization factor: 0.1305.

C \ W	5	50	100	500
0.01	∗ 0.9555 ±0.1930	0.9967 ±0.0053	0.9538 ±0.1927	0.9525 ±0.1924
0.10	∗ 0.9974 ±0.0079	0.9528 ±0.1923	0.9971 ±0.0031	0.8647 ±0.3151
0.50	∗ 0.9975 ±0.0079	∗ 0.9979 ±0.0029	0.9970 ±0.0024	0.9518 ±0.1921
5.00	**1.0000** ±0.0040	0.9535 ±0.1925	0.9093 ±0.2651	∗ 0.9106 ±0.2655
50.00	0.9969 ±0.0020	∗ 0.9983 ±0.0035	∗ 0.9101 ±0.2653	∗ 0.9538 ±0.1922

(b) Best candidate throughput (races/s) normalized to the fastest candidate. Normalization factor: 0.1363.

C \ W	5	50	100	500
0.01	∗ 1.0005 ±0.0023	∗ 1.0019 ±0.0032	∗ 1.0008 ±0.0022	1.0018 ±0.0039
0.10	∗ 1.0011 ±0.0031	∗ 1.0045 ±0.0132	∗ 1.0006 ±0.0021	∗ 1.0032 ±0.0096
0.50	∗ 1.0029 ±0.0133	∗ 1.0009 ±0.0027	∗ 1.0008 ±0.0023	∗ 1.0010 ±0.0026
5.00	∗ 1.0001 ±0.0009	1.0010 ±0.0020	∗ 1.0009 ±0.0021	∗ 1.0008 ±0.0021
50.00	1.0079 ±0.0198	∗ 1.0012 ±0.0025	**1.0000** ±0.0005	∗ 1.0026 ±0.0061

(c) Accuracy Root Mean Square Error (RMSE) normalized to the lowest one achieved. Normalization factor: 0.1015.

C \ W	5	50	100	500
0.01	**1.0000** ±0.0000	**1.0000** ±0.0000	**1.0000** ±0.0000	**1.0000** ±0.0000
0.10	**1.0000** ±0.0000	**1.0000** ±0.0000	**1.0000** ±0.0000	**1.0000** ±0.0000
0.50	**1.0000** ±0.0000	**1.0000** ±0.0000	**1.0000** ±0.0000	**1.0000** ±0.0000
5.00	**1.0000** ±0.0000	**1.0000** ±0.0000	**1.0000** ±0.0000	**1.0000** ±0.0000
50.00	**1.0000** ±0.0000	**1.0000** ±0.0000	**1.0000** ±0.0000	**1.0000** ±0.0000

(d) Percentage of races that met accuracy target normalized to the highest one achieved. Normalization factor: 100.0000.

C \ W	5	50	100	500
0.01	1.1628 ±0.4766	1.1163 ±0.2712	1.1163 ±0.3418	1.0698 ±0.2590
0.10	1.3256 ±0.6623	1.2558 ±0.5116	1.0698 ±0.2131	1.1163 ±0.4966
0.50	1.0930 ±0.2662	1.1395 ±0.4527	1.0930 ±0.2662	1.1163 ±0.5179
5.00	1.3023 ±0.6512	**1.0000** ±0.2218	1.1395 ±0.3442	1.0465 ±0.3247
50.00	1.0698 ±0.2590	1.0465 ±0.2494	1.1395 ±0.3743	1.1395 ±0.3743

(e) Time to convergence (number of races) normalized to the lowest one achieved. Normalization factor: 2.1500.

Figure 5-9: Metrics for benchmark Image Compression on the AMD48 system evaluated with different values of hyperparameters. An asterisk ∗ next to a number means that the difference from optimum is not statistically significant (p-value ≥ 0.05).

Chapter 6

Conclusions and Future Work

PetaBricks [4, 21, 7, 3, 5] is an implicitly parallel programming language which, though a process called *autotuning*, can automatically optimize programs for fast, QoS-aware execution on any hardware. In this thesis, we presented and evaluated two PetaBricks autotuners: INCREA and SiblingRivalry.

INCREA is an offline autotuner based on a novel evolutionary algorithm. The EA is designed for problems such as autotuning, which are suited to incremental shortcuts, and which require such shortcuts because of their large search spaces and expensive solution evaluation. Furthermore, INCREA efficiently handles other problems inherent in autotuning, such as noisy fitness evaluation. We found that INCREA achieved significant speedups on 3 benchmarks.

With the SiblingRivalry online autotuner, we demonstrated that it can sometimes be more effective to devote resources to learning the smart thing to do, than to simply throw resources at a potentially suboptimal configuration. Our technique devoted half of the system resources to a search for better configurations, enabling online adaption to the system environment. We demonstrated how, despite doing more work, SiblingRivalry actually increased the performance and improved power consumption of the tuned programs in the face of dynamically changing execution conditions. SiblingRivalry was also able to fully eliminate the offline learning step, making the process fully transparent to users, which is the biggest impediment to the acceptance of autotuning. By eliminating any extra steps, we believe that SiblingRivalry can

bring autotuning to the mainstream program optimization. As we keep increasing the core counts of our processors, autotuning via SiblingRivalry can help exploit them in a purposeful way.

We also investigated the influence of hyperparameters on SiblingRivalry's autotuning performance. We demonstrated that the optimal choice of hyperparameters differs between programs and autotuning scenarios, but there exist "sensible defaults" that perform well across many programs. These defaults eliminate the need, in most cases, for the user to manually tweak SiblingRivalry for each autotuned program, thus making our approach more feasible in a real-world setting.

Future work includes evaluating our autotuners on large, real-world applications, as opposed to single benchmarks presented in this thesis. We would also like to see SiblingRivalry applied in a cloud setting, where architecture migrations occur live and are not simulated. We believe such tests would reveal more benefits of program autotuning, as well as show areas where the autotuners could improve. Furthermore, we would like to make our autotuners more aware of the distinction between algorithmic choices and cutoffs/tunables. While these types of parameters have significantly different tuning characteristics, they are currently treated equally by the autotuners. Moreover, we would like to investigate online autotuning without racing, where all available cores can be devoted to the autotuned program. This is especially important as the number of multicore-aware, highly parallelizable programs increases.

Finally, we would like to completely eliminate the need for tweaking hyperparameters in SiblingRivalry, perhaps by tuning them automatically using a model or by dynamically adjusting them during the autotuner's execution. We especially expect the ability to dynamically adjust hyperparameters during a run to bring tangible performance benefits, as different exploitation/exploration trade-offs could be best at different points in the autotuning process.

Bibliography

[1] A. N. Aizawa and B. W. Wah. Scheduling of genetic algorithms in a noisy environment. *Evolutionary Computation*, 2(2):97–122, 1994.

[2] A. Ali, L. Johnsson, and J. Subhlok. Scheduling FFT computation on SMP and multicore systems. In *Supercomputing*, New York, NY, 2007.

[3] J. Ansel, Y. L. W. ans Cy Chan, M. Olszewski, A. Edelman, and S. Amarasinghe. Language and compiler support for auto-tuning variable-accuracy algorithms. In *CGO*, Chamonix, France, Apr 2011.

[4] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. PetaBricks: A language and compiler for algorithmic choice. In *PLDI*, Dublin, Ireland, Jun 2009.

[5] J. Ansel, M. Pacula, S. Amarasinghe, and U.-M. O'Reilly. An efficient evolutionary algorithm for solving bottom up problems. In *Annual Conference on Genetic and Evolutionary Computation*, Dublin, Ireland, July 2011.

[6] J. Ansel, M. Pacula, Y. L. Wong, C. Chan, M. Olszewski, U.-M. O'Reilly, and S. Amarasinghe. SiblingRivalry: Online autotuning through local competitions (under review).

[7] J. Ansel, Y. L. Won, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe. Language and compiler support for auto-tuning variable-accuracy algorithms. Technical Report MIT/CSAIL Technical Report MIT-CSAIL-TR-2010-032, Massachusetts Institute of Technology, Cambridge, MA, Jul 2010.

[8] D. V. Arnold and H.-G. Beyer. On the benefits of populations for noisy optimization. *Evolutionary Computation*, 11(2):111–127, 2003.

[9] D. Arthur and S. Vassilvitskii. k-means++: the advantages of careful seeding. In *SODA '07: Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[10] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire. The nonstochastic multiarmed bandit problem. *SIAM Journal on Computing*, 32(1), 2003.

[11] J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad. Fast, effective dynamic compilation. In *PLDI*, 1996.

[12] T. Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms.* Oxford University Press, New York NY, 1996.

[13] W. Baek and T. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *PLDI*, June 2010.

[14] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, S. Williams, and K. Yelick. Exascale computing study: Technology challenges in achieving exascale systems, 2008.

[15] V. Bhat, M. Parashar, . Hua Liu, M. Khandekar, N. Kandasamy, and S. Abdelwahed. Enabling self-managing applications using model-based online control strategies. In *International Conference on Autonomic Computing*, Washington, DC, 2006.

[16] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Supercomputing*, New York, NY, 1997.

[17] A. P. Bradley. The use of the area under the ROC curve in the evaluation of machine learning algorithms. *Pattern Recognition*, 30(7), 1997.

[18] J. Branke. Creating robust solutions by means of evolutionary algorithms. In A. Eiben, T. Baeck, M. Schoenauer, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature, PPSN V*, volume 1498 of *Lecture Notes in Computer Science*, pages 119–. Springer Berlin / Heidelberg, 1998.

[19] J. Branke, C. Schmidt, and H. Schmec. Efficient fitness estimation in noisy environments. In *Proceedings of Genetic and Evolutionary Computation*, pages 243–250, 2001.

[20] E. Cantu-Paz. Adaptive sampling for noisy problems. In *Genetic and Evolutionary Computation, GECCO 2004*, volume 3102 of *Lecture Notes in Computer Science*, pages 947–958. Springer Berlin / Heidelberg, 2004.

[21] C. Chan, J. Ansel, Y. L. Wong, S. Amarasinghe, and A. Edelman. Autotuning multigrid with PetaBricks. In *Supercomputing*, Portland, OR, Nov 2009.

[22] F. Chang and V. Karamcheti. A framework for automatic adaptation of tunable distributed applications. *Cluster Computing*, 4, March 2001.

[23] D. Chen, N. Vachharajani, R. Hundt, S.-w. Liao, V. Ramasamy, P. Yuan, W. Chen, and W. Zheng. Taming hardware event samples for FDO compilation. In *CGO*, New York, NY, 2010.

[24] L. DaCosta, A. Fialho, M. Schoenauer, and M. Sebag. Adaptive operator selection with dynamic multi-armed bandits. In *GECCO*, New York, NY, 2008.

[25] L. Davis. Adapting operator probabilities in genetic algorithms. In *ICGA*, San Francisco, CA, 1989.

[26] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6:182–197, 2002.

[27] P. C. Diniz and M. C. Rinard. Dynamic feedback: an effective technique for adaptive computing. In *PLDI*, New York, NY, 1997.

[28] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Springer-Verlag, 2003.

[29] A. Fialho. *Adaptive Operator Selection for Optimization*. PhD thesis, Université Paris-Sud XI, Orsay, France, December 2010.

[30] A. Fialho, R. Ros, M. Schoenauer, and M. Sebag. Comparison-based adaptive strategy selection with bandits in differential evolution. In R. S. et al., editor, *PPSN'10: Proc. 11th International Conference on Parallel Problem Solving from Nature*, volume 6238 of *LNCS*, pages 194–203. Springer, September 2010.

[31] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *IEEE International Conference on Acoustics Speech and Signal Processing*, volume 3, 1998.

[32] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *IEEE*, 93(2), February 2005. Invited paper, special issue on "Program Generation, Optimization, and Platform Adaptation".

[33] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal. Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments. In *ICAC*, New York, NY, 2010.

[34] H. Hoffmann, S. Misailovic, S. Sidiroglou, A. Agarwal, and M. Rinard. Using code perforation to improve performance, reduce energy consumption, and respond to failures. Technical Report MIT-CSAIL-TR-2209-042, Massachusetts Institute of Technology, Sep 2009.

[35] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Power-aware computing with dynamic knobs. In *ASPLOS*, 2011.

[36] E. Im and K. Yelick. Optimizing sparse matrix computations for register reuse in SPARSITY. In *International Conference on Computational Science*, 2001.

[37] K. A. D. Jong. *Evolutionary computation - a unified approach.* MIT Press, 2006.

[38] G. Karsai, A. Ledeczi, J. Sztipanovits, G. Peceli, G. Simon, and T. Kovacshazy. An approach to self-adaptive software based on supervisory control. In *International Workshop in Self-adaptive software*, 2001.

[39] X. Li, M. J. Garzarn, and D. Padua. Optimizing sorting with genetic algorithms. In *CGO*, 2005.

[40] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In L. M. L. Cam and J. Neyman, editors, *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of California Press, 1967.

[41] C. A. Markowski and E. P. Markowski. Conditions for the effectiveness of a preliminary test of variance. 1990.

[42] M. Mitchell. *An introduction to genetic algorithms.* MIT Press, 1998.

[43] M. Olszewski and M. Voss. Install-time system for automatic generation of optimized parallel sorting algorithms. In *PDPTA*, 2004.

[44] M. Püschel, J. M. F. Moura, B. Singer, J. Xiong, J. R. Johnson, D. A. Padua, M. M. Veloso, and R. W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing alogorithms. *IJHPCA*, 18(1), 2004.

[45] R. Schaefer, C. Cotta, J. Kolodziej, and G. Rudolph, editors. *Parallel Problem Solving from Nature*, volume 6238 of *Lecture Notes in Computer Science*, 2010.

[46] H. Takagi. Interactive evolutionary computation: fusion of the capabilities of EC optimization and human evaluation. In *Proceedings of the IEEE*, volume 89, pages 1275–1296, September 2001.

[47] A. Teller and D. Andre. Automatically choosing the number of fitness cases: The rational allocation of trials. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 321–328, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[48] D. Thierens. Adaptive strategies for operator allocation. In F. G. Lobo, C. F. Lima, and Z. Michalewicz, editors, *Parameter Setting in Evolutionary Algorithms*, volume 54 of *Studies in Computational Intelligence*. 2007.

[49] M. Voss and R. Eigenmann. Adapt: Automated de-coupled adaptive program transformation. In *International Conference on Parallel Processing*, 2000.

[50] M. Voss and R. Eigenmann. High-level adaptive program optimization with adapt. *ACM SIGPLAN Notices*, 36(7), 2001.

[51] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Scientific Discovery through Advanced Computing Conference*, Journal of Physics: Conference Series, San Francisco, CA, June 2005.

[52] P. Waldemar and T. Ramstad. Hybrid KLT-SVD image compression. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, Washington, DC, 1997.

[53] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Supercomputing*, Washington, DC, 1998.

[54] R. C. Whaley and A. Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2), February 2005.

Appendix A

Detailed Statistics

A.1 Hyperparameters Runs: Normality Testing

All normality tests were performed using the Anderson-Darling test from the Python SciPy library included with Ubuntu 11.04 (`scipy.stats.anderson`). In the tables below, `A2` is the test statistic, and the percentages correspond to `A2` thresholds for the given significance level. An `A2` value of `inf` means that the standard deviation of the samples was 0. The `NORMAL?` column specifies whether we can reject the normality hypothesis with 1% significance level.

A.1.1 Xeon8

Sort

Metric: Mean throughput (races/s) normalized to the fastest configuration.

C	W	NORMAL?	A2	15%	10%	5%	2.5%	1%
0.01	5	NORMALITY REJECTED	1.284314	0.521	0.593	0.712	0.830	0.988
0.01	50	NORMALITY REJECTED	1.228355	0.521	0.593	0.712	0.830	0.988
0.01	100	NORMALITY REJECTED	1.508193	0.521	0.593	0.712	0.830	0.988
0.01	500	NORMALITY REJECTED	1.560355	0.521	0.593	0.712	0.830	0.988
0.10	5	NORMALITY REJECTED	1.534702	0.521	0.593	0.712	0.830	0.988
0.10	50	NORMALITY REJECTED	2.100498	0.521	0.593	0.712	0.830	0.988
0.10	100	NORMALITY REJECTED	3.441434	0.521	0.593	0.712	0.830	0.988
0.10	500	MAY BE NORMAL	0.831769	0.521	0.593	0.712	0.830	0.988
0.50	5	NORMALITY REJECTED	1.368860	0.521	0.593	0.712	0.830	0.988
0.50	50	NORMALITY REJECTED	2.374129	0.521	0.593	0.712	0.830	0.988

C	W	NORMAL?	A2	15%	10%	5%	2.5%	1%
0.50	100	NORMALITY REJECTED	1.685954	0.521	0.593	0.712	0.830	0.988
0.50	500	NORMALITY REJECTED	1.787032	0.521	0.593	0.712	0.830	0.988
5.00	5	NORMALITY REJECTED	1.163137	0.521	0.593	0.712	0.830	0.988
5.00	50	NORMALITY REJECTED	3.058326	0.521	0.593	0.712	0.830	0.988
5.00	100	NORMALITY REJECTED	2.727096	0.521	0.593	0.712	0.830	0.988
5.00	500	NORMALITY REJECTED	3.071712	0.521	0.593	0.712	0.830	0.988
50.00	5	NORMALITY REJECTED	1.358510	0.521	0.593	0.712	0.830	0.988
50.00	50	MAY BE NORMAL	0.611514	0.521	0.593	0.712	0.830	0.988
50.00	100	MAY BE NORMAL	0.816525	0.521	0.593	0.712	0.830	0.988
50.00	500	NORMALITY REJECTED	1.030630	0.521	0.593	0.712	0.830	0.988

Metric: Best candidate throughput (races/s) normalized to the fastest candidate.

C	W	NORMAL?	A2	15%	10%	5%	2.5%	1%
0.01	5	NORMALITY REJECTED	1.101815	0.521	0.593	0.712	0.830	0.988
0.01	50	NORMALITY REJECTED	1.226720	0.521	0.593	0.712	0.830	0.988
0.01	100	NORMALITY REJECTED	1.463621	0.521	0.593	0.712	0.830	0.988
0.01	500	NORMALITY REJECTED	1.417161	0.521	0.593	0.712	0.830	0.988
0.10	5	NORMALITY REJECTED	1.620384	0.521	0.593	0.712	0.830	0.988
0.10	50	NORMALITY REJECTED	2.210501	0.521	0.593	0.712	0.830	0.988
0.10	100	NORMALITY REJECTED	3.945761	0.521	0.593	0.712	0.830	0.988
0.10	500	NORMALITY REJECTED	1.144354	0.521	0.593	0.712	0.830	0.988
0.50	5	MAY BE NORMAL	0.894749	0.521	0.593	0.712	0.830	0.988
0.50	50	NORMALITY REJECTED	2.409091	0.521	0.593	0.712	0.830	0.988
0.50	100	NORMALITY REJECTED	1.741366	0.521	0.593	0.712	0.830	0.988
0.50	500	NORMALITY REJECTED	2.360866	0.521	0.593	0.712	0.830	0.988
5.00	5	MAY BE NORMAL	0.962929	0.521	0.593	0.712	0.830	0.988
5.00	50	NORMALITY REJECTED	3.053767	0.521	0.593	0.712	0.830	0.988
5.00	100	NORMALITY REJECTED	2.622177	0.521	0.593	0.712	0.830	0.988
5.00	500	NORMALITY REJECTED	3.742517	0.521	0.593	0.712	0.830	0.988
50.00	5	NORMALITY REJECTED	1.309038	0.521	0.593	0.712	0.830	0.988
50.00	50	NORMALITY REJECTED	1.198756	0.521	0.593	0.712	0.830	0.988
50.00	100	MAY BE NORMAL	0.623742	0.521	0.593	0.712	0.830	0.988
50.00	500	MAY BE NORMAL	0.186557	0.521	0.593	0.712	0.830	0.988

Metric: Time to convergence (number of races) normalized to the lowest one achieved.

C	W	NORMAL?	A2	15%	10%	5%	2.5%	1%
0.01	5	NORMALITY REJECTED	1.244822	0.521	0.593	0.712	0.830	0.988
0.01	50	NORMALITY REJECTED	3.276607	0.521	0.593	0.712	0.830	0.988
0.01	100	MAY BE NORMAL	0.733010	0.521	0.593	0.712	0.830	0.988
0.01	500	NORMALITY REJECTED	1.526122	0.521	0.593	0.712	0.830	0.988
0.10	5	NORMALITY REJECTED	4.499189	0.521	0.593	0.712	0.830	0.988
0.10	50	NORMALITY REJECTED	1.587559	0.521	0.593	0.712	0.830	0.988
0.10	100	NORMALITY REJECTED	1.839714	0.521	0.593	0.712	0.830	0.988

0.10	500	NORMALITY REJECTED	1.355015	0.521	0.593	0.712	0.830	0.988
0.50	5	NORMALITY REJECTED	3.425307	0.521	0.593	0.712	0.830	0.988
0.50	50	NORMALITY REJECTED	1.728002	0.521	0.593	0.712	0.830	0.988
0.50	100	MAY BE NORMAL	0.922450	0.521	0.593	0.712	0.830	0.988
0.50	500	MAY BE NORMAL	0.833637	0.521	0.593	0.712	0.830	0.988
5.00	5	NORMALITY REJECTED	3.035571	0.521	0.593	0.712	0.830	0.988
5.00	50	NORMALITY REJECTED	1.093509	0.521	0.593	0.712	0.830	0.988
5.00	100	MAY BE NORMAL	0.838164	0.521	0.593	0.712	0.830	0.988
5.00	500	NORMALITY REJECTED	1.237738	0.521	0.593	0.712	0.830	0.988
50.00	5	MAY BE NORMAL	0.928063	0.521	0.593	0.712	0.830	0.988
50.00	50	NORMALITY REJECTED	1.386931	0.521	0.593	0.712	0.830	0.988
50.00	100	NORMALITY REJECTED	1.227667	0.521	0.593	0.712	0.830	0.988
50.00	500	NORMALITY REJECTED	2.445985	0.521	0.593	0.712	0.830	0.988

Matrix Approximation

Metric: Mean throughput (races/s) normalized to the fastest configuration.

C	W	NORMAL?	A2	15%	10%	5%	2.5%	1%
0.01	5	NORMALITY REJECTED	8.252833	0.521	0.593	0.712	0.830	0.988
0.01	50	NORMALITY REJECTED	7.713824	0.521	0.593	0.712	0.830	0.988
0.01	100	NORMALITY REJECTED	7.504382	0.521	0.593	0.712	0.830	0.988
0.01	500	MAY BE NORMAL	0.408905	0.521	0.593	0.712	0.830	0.988
0.10	5	NORMALITY REJECTED	6.578900	0.521	0.593	0.712	0.830	0.988
0.10	50	NORMALITY REJECTED	6.729593	0.521	0.593	0.712	0.830	0.988
0.10	100	NORMALITY REJECTED	2.160410	0.521	0.593	0.712	0.830	0.988
0.10	500	NORMALITY REJECTED	8.649129	0.521	0.593	0.712	0.830	0.988
0.50	5	MAY BE NORMAL	0.279776	0.521	0.593	0.712	0.830	0.988
0.50	50	NORMALITY REJECTED	8.521536	0.521	0.593	0.712	0.830	0.988
0.50	100	NORMALITY REJECTED	7.852097	0.521	0.593	0.712	0.830	0.988
0.50	500	NORMALITY REJECTED	7.325877	0.521	0.593	0.712	0.830	0.988
5.00	5	NORMALITY REJECTED	8.844623	0.521	0.593	0.712	0.830	0.988
5.00	50	NORMALITY REJECTED	8.254022	0.521	0.593	0.712	0.830	0.988
5.00	100	NORMALITY REJECTED	7.607129	0.521	0.593	0.712	0.830	0.988
5.00	500	NORMALITY REJECTED	8.182801	0.521	0.593	0.712	0.830	0.988
50.00	5	NORMALITY REJECTED	8.613488	0.521	0.593	0.712	0.830	0.988
50.00	50	NORMALITY REJECTED	8.376068	0.521	0.593	0.712	0.830	0.988
50.00	100	NORMALITY REJECTED	8.905095	0.521	0.593	0.712	0.830	0.988
50.00	500	NORMALITY REJECTED	8.834549	0.521	0.593	0.712	0.830	0.988

Metric: Best candidate throughput (races/s) normalized to the fastest candidate.

C	W	NORMAL?	A2	15%	10%	5%	2.5%	1%
0.01	5	NORMALITY REJECTED	8.852926	0.521	0.593	0.712	0.830	0.988

0.01	50	NORMALITY REJECTED	8.661892	0.521	0.593	0.712	0.830	0.988
0.01	100	NORMALITY REJECTED	8.161498	0.521	0.593	0.712	0.830	0.988
0.01	500	NORMALITY REJECTED	1.423275	0.521	0.593	0.712	0.830	0.988
0.10	5	NORMALITY REJECTED	6.868272	0.521	0.593	0.712	0.830	0.988
0.10	50	NORMALITY REJECTED	7.231154	0.521	0.593	0.712	0.830	0.988
0.10	100	NORMALITY REJECTED	1.003293	0.521	0.593	0.712	0.830	0.988
0.10	500	NORMALITY REJECTED	9.095262	0.521	0.593	0.712	0.830	0.988
0.50	5	MAY BE NORMAL	0.607329	0.521	0.593	0.712	0.830	0.988
0.50	50	NORMALITY REJECTED	8.946796	0.521	0.593	0.712	0.830	0.988
0.50	100	NORMALITY REJECTED	9.098551	0.521	0.593	0.712	0.830	0.988
0.50	500	NORMALITY REJECTED	7.641659	0.521	0.593	0.712	0.830	0.988
5.00	5	NORMALITY REJECTED	9.245544	0.521	0.593	0.712	0.830	0.988
5.00	50	NORMALITY REJECTED	8.523992	0.521	0.593	0.712	0.830	0.988
5.00	100	NORMALITY REJECTED	7.835793	0.521	0.593	0.712	0.830	0.988
5.00	500	NORMALITY REJECTED	8.290691	0.521	0.593	0.712	0.830	0.988
50.00	5	NORMALITY REJECTED	8.727063	0.521	0.593	0.712	0.830	0.988
50.00	50	NORMALITY REJECTED	9.108300	0.521	0.593	0.712	0.830	0.988
50.00	100	NORMALITY REJECTED	9.111395	0.521	0.593	0.712	0.830	0.988
50.00	500	NORMALITY REJECTED	8.918689	0.521	0.593	0.712	0.830	0.988

Metric: Accuracy Root Mean Square Error (RMSE) normalized to the lowest one achieved.

C	W	NORMAL?	A2	15%	10%	5%	2.5%	1%
0.01	5	NORMALITY REJECTED	5.349325	0.521	0.593	0.712	0.830	0.988
0.01	50	NORMALITY REJECTED	4.752627	0.521	0.593	0.712	0.830	0.988
0.01	100	NORMALITY REJECTED	8.764931	0.521	0.593	0.712	0.830	0.988
0.01	500	NORMALITY REJECTED	3.895148	0.521	0.593	0.712	0.830	0.988
0.10	5	NORMALITY REJECTED	4.482589	0.521	0.593	0.712	0.830	0.988
0.10	50	NORMALITY REJECTED	4.869248	0.521	0.593	0.712	0.830	0.988
0.10	100	NORMALITY REJECTED	4.760775	0.521	0.593	0.712	0.830	0.988
0.10	500	NORMALITY REJECTED	4.628927	0.521	0.593	0.712	0.830	0.988
0.50	5	NORMALITY REJECTED	3.703395	0.521	0.593	0.712	0.830	0.988
0.50	50	NORMALITY REJECTED	2.936389	0.521	0.593	0.712	0.830	0.988
0.50	100	NORMALITY REJECTED	6.072103	0.521	0.593	0.712	0.830	0.988
0.50	500	NORMALITY REJECTED	5.275254	0.521	0.593	0.712	0.830	0.988
5.00	5	NORMALITY REJECTED	4.625427	0.521	0.593	0.712	0.830	0.988
5.00	50	NORMALITY REJECTED	1.866134	0.521	0.593	0.712	0.830	0.988
5.00	100	NORMALITY REJECTED	3.763328	0.521	0.593	0.712	0.830	0.988
5.00	500	NORMALITY REJECTED	3.258102	0.521	0.593	0.712	0.830	0.988
50.00	5	NORMALITY REJECTED	2.549410	0.521	0.593	0.712	0.830	0.988
50.00	50	NORMALITY REJECTED	2.586148	0.521	0.593	0.712	0.830	0.988
50.00	100	NORMALITY REJECTED	2.447889	0.521	0.593	0.712	0.830	0.988
50.00	500	NORMALITY REJECTED	2.437531	0.521	0.593	0.712	0.830	0.988

Metric: Percentage of races that met accuracy target normalized to the highest one achieved.

C	W	NORMAL?	A2	15%	10%	5%	2.5%	1%
0.01	5	NORMALITY REJECTED	inf	0.521	0.593	0.712	0.830	0.988
0.01	50	NORMALITY REJECTED	inf	0.521	0.593	0.712	0.830	0.988
0.01	100	NORMALITY REJECTED	inf	0.521	0.593	0.712	0.830	0.988
0.01	500	NORMALITY REJECTED	inf	0.521	0.593	0.712	0.830	0.988
0.10	5	NORMALITY REJECTED	inf	0.521	0.593	0.712	0.830	0.988
0.10	50	NORMALITY REJECTED	inf	0.521	0.593	0.712	0.830	0.988
0.10	100	NORMALITY REJECTED	inf	0.521	0.593	0.712	0.830	0.988
0.10	500	NORMALITY REJECTED	inf	0.521	0.593	0.712	0.830	0.988
0.50	5	NORMALITY REJECTED	inf	0.521	0.593	0.712	0.830	0.988
0.50	50	NORMALITY REJECTED	inf	0.521	0.593	0.712	0.830	0.988
0.50	100	NORMALITY REJECTED	inf	0.521	0.593	0.712	0.830	0.988
0.50	500	NORMALITY REJECTED	inf	0.521	0.593	0.712	0.830	0.988
5.00	5	NORMALITY REJECTED	inf	0.521	0.593	0.712	0.830	0.988
5.00	50	NORMALITY REJECTED	inf	0.521	0.593	0.712	0.830	0.988
5.00	100	NORMALITY REJECTED	inf	0.521	0.593	0.712	0.830	0.988
5.00	500	NORMALITY REJECTED	inf	0.521	0.593	0.712	0.830	0.988
50.00	5	NORMALITY REJECTED	inf	0.521	0.593	0.712	0.830	0.988
50.00	50	NORMALITY REJECTED	inf	0.521	0.593	0.712	0.830	0.988
50.00	100	NORMALITY REJECTED	inf	0.521	0.593	0.712	0.830	0.988
50.00	500	NORMALITY REJECTED	inf	0.521	0.593	0.712	0.830	0.988

Metric: Time to convergence (number of races) normalized to the lowest one achieved.

C	W	NORMAL?	A2	15%	10%	5%	2.5%	1%
0.01	5	NORMALITY REJECTED	1.788978	0.521	0.593	0.712	0.830	0.988
0.01	50	NORMALITY REJECTED	2.816785	0.521	0.593	0.712	0.830	0.988
0.01	100	NORMALITY REJECTED	1.014054	0.521	0.593	0.712	0.830	0.988
0.01	500	NORMALITY REJECTED	1.360587	0.521	0.593	0.712	0.830	0.988
0.10	5	NORMALITY REJECTED	1.305490	0.521	0.593	0.712	0.830	0.988
0.10	50	NORMALITY REJECTED	2.299307	0.521	0.593	0.712	0.830	0.988
0.10	100	NORMALITY REJECTED	2.046253	0.521	0.593	0.712	0.830	0.988
0.10	500	NORMALITY REJECTED	2.863391	0.521	0.593	0.712	0.830	0.988
0.50	5	NORMALITY REJECTED	5.122732	0.521	0.593	0.712	0.830	0.988
0.50	50	NORMALITY REJECTED	1.208050	0.521	0.593	0.712	0.830	0.988
0.50	100	NORMALITY REJECTED	2.011122	0.521	0.593	0.712	0.830	0.988
0.50	500	NORMALITY REJECTED	1.189583	0.521	0.593	0.712	0.830	0.988
5.00	5	NORMALITY REJECTED	2.115248	0.521	0.593	0.712	0.830	0.988
5.00	50	NORMALITY REJECTED	2.059417	0.521	0.593	0.712	0.830	0.988
5.00	100	NORMALITY REJECTED	1.093626	0.521	0.593	0.712	0.830	0.988
5.00	500	NORMALITY REJECTED	1.138205	0.521	0.593	0.712	0.830	0.988
50.00	5	NORMALITY REJECTED	1.723931	0.521	0.593	0.712	0.830	0.988
50.00	50	NORMALITY REJECTED	1.318870	0.521	0.593	0.712	0.830	0.988
50.00	100	NORMALITY REJECTED	3.591802	0.521	0.593	0.712	0.830	0.988

50.00	500	NORMALITY REJECTED	7.677452	0.521	0.593	0.712	0.830	0.988

Poisson

Metric: Mean throughput (races/s) normalized to the fastest configuration.

C	W	NORMAL?	A2	15%	10%	5%	2.5%	1%
0.01	5	NORMALITY REJECTED	5.063804	0.521	0.593	0.712	0.830	0.988
0.01	50	NORMALITY REJECTED	2.456555	0.521	0.593	0.712	0.830	0.988
0.01	100	NORMALITY REJECTED	2.873244	0.521	0.593	0.712	0.830	0.988
0.01	500	NORMALITY REJECTED	2.529990	0.521	0.593	0.712	0.830	0.988
0.10	5	NORMALITY REJECTED	4.291440	0.521	0.593	0.712	0.830	0.988
0.10	50	NORMALITY REJECTED	9.710037	0.521	0.593	0.712	0.830	0.988
0.10	100	NORMALITY REJECTED	3.451451	0.521	0.593	0.712	0.830	0.988
0.10	500	NORMALITY REJECTED	4.357392	0.521	0.593	0.712	0.830	0.988
0.50	5	NORMALITY REJECTED	4.205766	0.521	0.593	0.712	0.830	0.988
0.50	50	NORMALITY REJECTED	6.329573	0.521	0.593	0.712	0.830	0.988
0.50	100	NORMALITY REJECTED	6.970498	0.521	0.593	0.712	0.830	0.988
0.50	500	NORMALITY REJECTED	6.034668	0.521	0.593	0.712	0.830	0.988
5.00	5	NORMALITY REJECTED	3.649729	0.521	0.593	0.712	0.830	0.988
5.00	50	NORMALITY REJECTED	9.968898	0.521	0.593	0.712	0.830	0.988
5.00	100	NORMALITY REJECTED	10.040293	0.521	0.593	0.712	0.830	0.988
5.00	500	NORMALITY REJECTED	8.355336	0.521	0.593	0.712	0.830	0.988
50.00	5	NORMALITY REJECTED	6.073488	0.521	0.593	0.712	0.830	0.988
50.00	50	NORMALITY REJECTED	4.246614	0.521	0.593	0.712	0.830	0.988
50.00	100	NORMALITY REJECTED	4.374201	0.521	0.593	0.712	0.830	0.988
50.00	500	NORMALITY REJECTED	2.012314	0.521	0.593	0.712	0.830	0.988

Metric: Best candidate throughput (races/s) normalized to the fastest candidate.

C	W	NORMAL?	A2	15%	10%	5%	2.5%	1%
0.01	5	NORMALITY REJECTED	1.983814	0.521	0.593	0.712	0.830	0.988
0.01	50	MAY BE NORMAL	0.385748	0.521	0.593	0.712	0.830	0.988
0.01	100	MAY BE NORMAL	0.865270	0.521	0.593	0.712	0.830	0.988
0.01	500	MAY BE NORMAL	0.820785	0.521	0.593	0.712	0.830	0.988
0.10	5	NORMALITY REJECTED	1.942145	0.521	0.593	0.712	0.830	0.988
0.10	50	NORMALITY REJECTED	2.982168	0.521	0.593	0.712	0.830	0.988
0.10	100	NORMALITY REJECTED	10.973696	0.521	0.593	0.712	0.830	0.988
0.10	500	NORMALITY REJECTED	10.020671	0.521	0.593	0.712	0.830	0.988
0.50	5	NORMALITY REJECTED	1.092976	0.521	0.593	0.712	0.830	0.988
0.50	50	MAY BE NORMAL	0.760534	0.521	0.593	0.712	0.830	0.988
0.50	100	NORMALITY REJECTED	1.192934	0.521	0.593	0.712	0.830	0.988
0.50	500	NORMALITY REJECTED	9.375558	0.521	0.593	0.712	0.830	0.988
5.00	5	NORMALITY REJECTED	2.065992	0.521	0.593	0.712	0.830	0.988

5.00	50	NORMALITY REJECTED	1.853904	0.521	0.593	0.712	0.830	0.988
5.00	100	NORMALITY REJECTED	1.516659	0.521	0.593	0.712	0.830	0.988
5.00	500	NORMALITY REJECTED	11.039467	0.521	0.593	0.712	0.830	0.988
50.00	5	NORMALITY REJECTED	9.853828	0.521	0.593	0.712	0.830	0.988
50.00	50	NORMALITY REJECTED	8.411827	0.521	0.593	0.712	0.830	0.988
50.00	100	NORMALITY REJECTED	8.530623	0.521	0.593	0.712	0.830	0.988
50.00	500	NORMALITY REJECTED	3.230526	0.521	0.593	0.712	0.830	0.988

Metric: Accuracy Root Mean Square Error (RMSE) normalized to the lowest one achieved.

C	W	NORMAL?	A2	15%	10%	5%	2.5%	1%
0.01	5	NORMALITY REJECTED	11.090074	0.521	0.593	0.712	0.830	0.988
0.01	50	NORMALITY REJECTED	2.747284	0.521	0.593	0.712	0.830	0.988
0.01	100	NORMALITY REJECTED	2.971172	0.521	0.593	0.712	0.830	0.988
0.01	500	NORMALITY REJECTED	2.487303	0.521	0.593	0.712	0.830	0.988
0.10	5	NORMALITY REJECTED	10.467762	0.521	0.593	0.712	0.830	0.988
0.10	50	NORMALITY REJECTED	9.722765	0.521	0.593	0.712	0.830	0.988
0.10	100	NORMALITY REJECTED	10.950510	0.521	0.593	0.712	0.830	0.988
0.10	500	NORMALITY REJECTED	9.854671	0.521	0.593	0.712	0.830	0.988
0.50	5	NORMALITY REJECTED	11.090074	0.521	0.593	0.712	0.830	0.988
0.50	50	NORMALITY REJECTED	6.413800	0.521	0.593	0.712	0.830	0.988
0.50	100	NORMALITY REJECTED	7.437294	0.521	0.593	0.712	0.830	0.988
0.50	500	NORMALITY REJECTED	9.357809	0.521	0.593	0.712	0.830	0.988
5.00	5	NORMALITY REJECTED	10.465672	0.521	0.593	0.712	0.830	0.988
5.00	50	NORMALITY REJECTED	10.104409	0.521	0.593	0.712	0.830	0.988
5.00	100	NORMALITY REJECTED	10.432942	0.521	0.593	0.712	0.830	0.988
5.00	500	NORMALITY REJECTED	10.656293	0.521	0.593	0.712	0.830	0.988
50.00	5	NORMALITY REJECTED	5.337664	0.521	0.593	0.712	0.830	0.988
50.00	50	NORMALITY REJECTED	10.164875	0.521	0.593	0.712	0.830	0.988
50.00	100	NORMALITY REJECTED	7.524575	0.521	0.593	0.712	0.830	0.988
50.00	500	NORMALITY REJECTED	3.103251	0.521	0.593	0.712	0.830	0.988

Metric: Percentage of races that met accuracy target normalized to the highest one achieved.

C	W	NORMAL?	A2	15%	10%	5%	2.5%	1%
0.01	5	NORMALITY REJECTED	11.090074	0.521	0.593	0.712	0.830	0.988
0.01	50	NORMALITY REJECTED	2.738447	0.521	0.593	0.712	0.830	0.988
0.01	100	NORMALITY REJECTED	2.971044	0.521	0.593	0.712	0.830	0.988
0.01	500	NORMALITY REJECTED	2.477289	0.521	0.593	0.712	0.830	0.988
0.10	5	NORMALITY REJECTED	10.467762	0.521	0.593	0.712	0.830	0.988
0.10	50	NORMALITY REJECTED	9.709353	0.521	0.593	0.712	0.830	0.988
0.10	100	NORMALITY REJECTED	3.737506	0.521	0.593	0.712	0.830	0.988
0.10	500	NORMALITY REJECTED	5.100753	0.521	0.593	0.712	0.830	0.988
0.50	5	NORMALITY REJECTED	11.090074	0.521	0.593	0.712	0.830	0.988
0.50	50	NORMALITY REJECTED	6.413796	0.521	0.593	0.712	0.830	0.988

0.50	100	NORMALITY REJECTED	7.427301	0.521	0.593	0.712	0.830	0.988
0.50	500	NORMALITY REJECTED	7.030439	0.521	0.593	0.712	0.830	0.988
5.00	5	NORMALITY REJECTED	10.465672	0.521	0.593	0.712	0.830	0.988
5.00	50	NORMALITY REJECTED	10.096426	0.521	0.593	0.712	0.830	0.988
5.00	100	NORMALITY REJECTED	10.416650	0.521	0.593	0.712	0.830	0.988
5.00	500	NORMALITY REJECTED	9.184163	0.521	0.593	0.712	0.830	0.988
50.00	5	NORMALITY REJECTED	5.775978	0.521	0.593	0.712	0.830	0.988
50.00	50	NORMALITY REJECTED	3.006555	0.521	0.593	0.712	0.830	0.988
50.00	100	NORMALITY REJECTED	4.863690	0.521	0.593	0.712	0.830	0.988
50.00	500	NORMALITY REJECTED	3.221718	0.521	0.593	0.712	0.830	0.988

Metric: Time to convergence (number of races) normalized to the lowest one achieved.

C	W	NORMAL?	A2	15%	10%	5%	2.5%	1%
0.01	5	NORMALITY REJECTED	4.104363	0.521	0.593	0.712	0.830	0.988
0.01	50	NORMALITY REJECTED	4.512714	0.521	0.593	0.712	0.830	0.988
0.01	100	NORMALITY REJECTED	5.743497	0.521	0.593	0.712	0.830	0.988
0.01	500	NORMALITY REJECTED	3.429277	0.521	0.593	0.712	0.830	0.988
0.10	5	NORMALITY REJECTED	4.407744	0.521	0.593	0.712	0.830	0.988
0.10	50	NORMALITY REJECTED	3.626448	0.521	0.593	0.712	0.830	0.988
0.10	100	NORMALITY REJECTED	11.019354	0.521	0.593	0.712	0.830	0.988
0.10	500	NORMALITY REJECTED	9.983349	0.521	0.593	0.712	0.830	0.988
0.50	5	NORMALITY REJECTED	5.477307	0.521	0.593	0.712	0.830	0.988
0.50	50	NORMALITY REJECTED	5.092620	0.521	0.593	0.712	0.830	0.988
0.50	100	NORMALITY REJECTED	4.096573	0.521	0.593	0.712	0.830	0.988
0.50	500	NORMALITY REJECTED	9.314761	0.521	0.593	0.712	0.830	0.988
5.00	5	NORMALITY REJECTED	6.035728	0.521	0.593	0.712	0.830	0.988
5.00	50	NORMALITY REJECTED	2.890074	0.521	0.593	0.712	0.830	0.988
5.00	100	NORMALITY REJECTED	7.306340	0.521	0.593	0.712	0.830	0.988
5.00	500	NORMALITY REJECTED	11.076543	0.521	0.593	0.712	0.830	0.988
50.00	5	NORMALITY REJECTED	10.741350	0.521	0.593	0.712	0.830	0.988
50.00	50	NORMALITY REJECTED	8.904896	0.521	0.593	0.712	0.830	0.988
50.00	100	NORMALITY REJECTED	8.159161	0.521	0.593	0.712	0.830	0.988
50.00	500	NORMALITY REJECTED	1.986158	0.521	0.593	0.712	0.830	0.988

Binpacking

Metric: Mean throughput (races/s) normalized to the fastest configuration.

C	W	NORMAL?	A2	15%	10%	5%	2.5%	1%
0.01	5	MAY BE NORMAL	0.511441	0.521	0.593	0.712	0.830	0.988
0.01	50	MAY BE NORMAL	0.899195	0.521	0.593	0.712	0.830	0.988
0.01	100	NORMALITY REJECTED	1.910947	0.521	0.593	0.712	0.830	0.988
0.01	500	NORMALITY REJECTED	1.105331	0.521	0.593	0.712	0.830	0.988

0.10	5	NORMALITY REJECTED	1.213818	0.521	0.593	0.712	0.830	0.988
0.10	50	NORMALITY REJECTED	4.525793	0.521	0.593	0.712	0.830	0.988
0.10	100	NORMALITY REJECTED	1.196611	0.521	0.593	0.712	0.830	0.988
0.10	500	NORMALITY REJECTED	1.400306	0.521	0.593	0.712	0.830	0.988
0.50	5	MAY BE NORMAL	0.530477	0.521	0.593	0.712	0.830	0.988
0.50	50	NORMALITY REJECTED	6.633089	0.521	0.593	0.712	0.830	0.988
0.50	100	NORMALITY REJECTED	1.327616	0.521	0.593	0.712	0.830	0.988
0.50	500	NORMALITY REJECTED	3.863110	0.521	0.593	0.712	0.830	0.988
5.00	5	NORMALITY REJECTED	1.369369	0.521	0.593	0.712	0.830	0.988
5.00	50	NORMALITY REJECTED	2.708605	0.521	0.593	0.712	0.830	0.988
5.00	100	NORMALITY REJECTED	2.988649	0.521	0.593	0.712	0.830	0.988
5.00	500	NORMALITY REJECTED	4.452346	0.521	0.593	0.712	0.830	0.988
50.00	5	MAY BE NORMAL	0.710471	0.521	0.593	0.712	0.830	0.988
50.00	50	NORMALITY REJECTED	3.563654	0.521	0.593	0.712	0.830	0.988
50.00	100	NORMALITY REJECTED	3.183385	0.521	0.593	0.712	0.830	0.988
50.00	500	MAY BE NORMAL	0.526345	0.521	0.593	0.712	0.830	0.988

Metric: Best candidate throughput (races/s) normalized to the fastest candidate.

C	W	NORMAL?	A2	15%	10%	5%	2.5%	1%
0.01	5	MAY BE NORMAL	0.525013	0.519	0.591	0.709	0.827	0.984
0.01	50	NORMALITY REJECTED	5.716674	0.521	0.593	0.712	0.830	0.988
0.01	100	NORMALITY REJECTED	7.112779	0.520	0.592	0.710	0.828	0.985
0.01	500	NORMALITY REJECTED	3.015718	0.521	0.593	0.712	0.830	0.988
0.10	5	NORMALITY REJECTED	2.879481	0.497	0.566	0.680	0.793	0.943
0.10	50	NORMALITY REJECTED	7.055436	0.521	0.593	0.712	0.830	0.988
0.10	100	NORMALITY REJECTED	7.802653	0.521	0.593	0.712	0.830	0.988
0.10	500	NORMALITY REJECTED	3.383823	0.521	0.593	0.712	0.830	0.988
0.50	5	NORMALITY REJECTED	1.782785	0.501	0.570	0.684	0.798	0.950
0.50	50	NORMALITY REJECTED	7.479624	0.520	0.592	0.710	0.828	0.985
0.50	100	NORMALITY REJECTED	3.149358	0.518	0.590	0.708	0.826	0.983
0.50	500	NORMALITY REJECTED	2.244974	0.518	0.590	0.708	0.826	0.983
5.00	5	NORMALITY REJECTED	1.351623	0.501	0.570	0.684	0.798	0.950
5.00	50	NORMALITY REJECTED	4.950733	0.520	0.592	0.710	0.828	0.985
5.00	100	NORMALITY REJECTED	6.225737	0.521	0.593	0.712	0.830	0.988
5.00	500	NORMALITY REJECTED	3.524882	0.518	0.590	0.708	0.826	0.983
50.00	5	NORMALITY REJECTED	4.906516	0.518	0.590	0.708	0.826	0.983
50.00	50	NORMALITY REJECTED	10.153597	0.521	0.593	0.712	0.830	0.988
50.00	100	MAY BE NORMAL	0.308988	0.521	0.593	0.712	0.830	0.988
50.00	500	MAY BE NORMAL	0.615155	0.521	0.593	0.712	0.830	0.988

Metric: Accuracy Root Mean Square Error (RMSE) normalized to the lowest one achieved.

C	W	NORMAL?	A2	15%	10%	5%	2.5%	1%
0.01	5	NORMALITY REJECTED	3.866812	0.521	0.593	0.712	0.830	0.988

0.01	50	NORMALITY REJECTED	1.741488	0.521	0.593	0.712	0.830	0.988
0.01	100	NORMALITY REJECTED	1.933980	0.521	0.593	0.712	0.830	0.988
0.01	500	NORMALITY REJECTED	1.337884	0.521	0.593	0.712	0.830	0.988
0.10	5	NORMALITY REJECTED	3.665543	0.521	0.593	0.712	0.830	0.988
0.10	50	NORMALITY REJECTED	1.163981	0.521	0.593	0.712	0.830	0.988
0.10	100	NORMALITY REJECTED	1.168991	0.521	0.593	0.712	0.830	0.988
0.10	500	NORMALITY REJECTED	1.274428	0.521	0.593	0.712	0.830	0.988
0.50	5	NORMALITY REJECTED	3.173595	0.521	0.593	0.712	0.830	0.988
0.50	50	MAY BE NORMAL	0.884278	0.521	0.593	0.712	0.830	0.988
0.50	100	NORMALITY REJECTED	0.997943	0.521	0.593	0.712	0.830	0.988
0.50	500	MAY BE NORMAL	0.680962	0.521	0.593	0.712	0.830	0.988
5.00	5	NORMALITY REJECTED	3.092460	0.521	0.593	0.712	0.830	0.988
5.00	50	MAY BE NORMAL	0.816055	0.521	0.593	0.712	0.830	0.988
5.00	100	NORMALITY REJECTED	1.259281	0.521	0.593	0.712	0.830	0.988
5.00	500	NORMALITY REJECTED	1.668342	0.521	0.593	0.712	0.830	0.988
50.00	5	MAY BE NORMAL	0.986628	0.521	0.593	0.712	0.830	0.988
50.00	50	MAY BE NORMAL	0.552315	0.521	0.593	0.712	0.830	0.988
50.00	100	NORMALITY REJECTED	1.140900	0.521	0.593	0.712	0.830	0.988
50.00	500	MAY BE NORMAL	0.689430	0.521	0.593	0.712	0.830	0.988

Metric: Percentage of races that met accuracy target normalized to the highest one achieved.

C	W	NORMAL?	A2	15%	10%	5%	2.5%	1%
0.01	5	NORMALITY REJECTED	6.486885	0.521	0.593	0.712	0.830	0.988
0.01	50	NORMALITY REJECTED	1.117620	0.521	0.593	0.712	0.830	0.988
0.01	100	MAY BE NORMAL	0.572489	0.521	0.593	0.712	0.830	0.988
0.01	500	MAY BE NORMAL	0.702000	0.521	0.593	0.712	0.830	0.988
0.10	5	NORMALITY REJECTED	4.150259	0.521	0.593	0.712	0.830	0.988
0.10	50	MAY BE NORMAL	0.597753	0.521	0.593	0.712	0.830	0.988
0.10	100	MAY BE NORMAL	0.860399	0.521	0.593	0.712	0.830	0.988
0.10	500	MAY BE NORMAL	0.514595	0.521	0.593	0.712	0.830	0.988
0.50	5	NORMALITY REJECTED	5.448775	0.521	0.593	0.712	0.830	0.988
0.50	50	MAY BE NORMAL	0.871365	0.521	0.593	0.712	0.830	0.988
0.50	100	NORMALITY REJECTED	1.227693	0.521	0.593	0.712	0.830	0.988
0.50	500	NORMALITY REJECTED	1.097267	0.521	0.593	0.712	0.830	0.988
5.00	5	NORMALITY REJECTED	5.525108	0.521	0.593	0.712	0.830	0.988
5.00	50	MAY BE NORMAL	0.709451	0.521	0.593	0.712	0.830	0.988
5.00	100	MAY BE NORMAL	0.644917	0.521	0.593	0.712	0.830	0.988
5.00	500	NORMALITY REJECTED	2.025630	0.521	0.593	0.712	0.830	0.988
50.00	5	NORMALITY REJECTED	3.506765	0.521	0.593	0.712	0.830	0.988
50.00	50	MAY BE NORMAL	0.308338	0.521	0.593	0.712	0.830	0.988
50.00	100	MAY BE NORMAL	0.479827	0.521	0.593	0.712	0.830	0.988
50.00	500	MAY BE NORMAL	0.257110	0.521	0.593	0.712	0.830	0.988

Metric: Time to convergence (number of races) normalized to the lowest one achieved.

C	W	NORMAL?	A2	15%	10%	5%	2.5%	1%
0.01	5	MAY BE NORMAL	0.424494	0.519	0.591	0.709	0.827	0.984
0.01	50	NORMALITY REJECTED	4.299974	0.521	0.593	0.712	0.830	0.988
0.01	100	NORMALITY REJECTED	5.615916	0.520	0.592	0.710	0.828	0.985
0.01	500	NORMALITY REJECTED	5.818405	0.521	0.593	0.712	0.830	0.988
0.10	5	NORMALITY REJECTED	0.995555	0.497	0.566	0.680	0.793	0.943
0.10	50	NORMALITY REJECTED	4.422117	0.521	0.593	0.712	0.830	0.988
0.10	100	NORMALITY REJECTED	5.958142	0.521	0.593	0.712	0.830	0.988
0.10	500	NORMALITY REJECTED	4.876148	0.521	0.593	0.712	0.830	0.988
0.50	5	MAY BE NORMAL	0.276588	0.501	0.570	0.684	0.798	0.950
0.50	50	NORMALITY REJECTED	9.940640	0.520	0.592	0.710	0.828	0.985
0.50	100	MAY BE NORMAL	0.900687	0.518	0.590	0.708	0.826	0.983
0.50	500	NORMALITY REJECTED	8.029892	0.518	0.590	0.708	0.826	0.983
5.00	5	MAY BE NORMAL	0.423961	0.501	0.570	0.684	0.798	0.950
5.00	50	NORMALITY REJECTED	3.468877	0.520	0.592	0.710	0.828	0.985
5.00	100	NORMALITY REJECTED	4.899663	0.521	0.593	0.712	0.830	0.988
5.00	500	NORMALITY REJECTED	2.451779	0.518	0.590	0.708	0.826	0.983
50.00	5	MAY BE NORMAL	0.956576	0.518	0.590	0.708	0.826	0.983
50.00	50	NORMALITY REJECTED	3.017714	0.521	0.593	0.712	0.830	0.988
50.00	100	NORMALITY REJECTED	1.754383	0.521	0.593	0.712	0.830	0.988
50.00	500	NORMALITY REJECTED	1.910622	0.521	0.593	0.712	0.830	0.988

A.1.2 AMD48

Sort

Metric: Mean throughput (races/s) normalized to the fastest configuration.

C	W	NORMAL?	A2	15%	10%	5%	2.5%	1%
0.01	5	NORMALITY REJECTED	1.089637	0.506	0.577	0.692	0.807	0.960
0.01	50	NORMALITY REJECTED	1.651235	0.506	0.577	0.692	0.807	0.960
0.01	100	NORMALITY REJECTED	1.411876	0.506	0.577	0.692	0.807	0.960
0.01	500	NORMALITY REJECTED	2.021908	0.506	0.577	0.692	0.807	0.960
0.10	5	MAY BE NORMAL	0.563654	0.506	0.577	0.692	0.807	0.960
0.10	50	NORMALITY REJECTED	2.502121	0.506	0.577	0.692	0.807	0.960
0.10	100	NORMALITY REJECTED	1.726439	0.506	0.577	0.692	0.807	0.960
0.10	500	NORMALITY REJECTED	2.339012	0.506	0.577	0.692	0.807	0.960
0.50	5	NORMALITY REJECTED	1.147831	0.506	0.577	0.692	0.807	0.960
0.50	50	NORMALITY REJECTED	1.185928	0.506	0.577	0.692	0.807	0.960
0.50	100	NORMALITY REJECTED	2.281348	0.506	0.577	0.692	0.807	0.960
0.50	500	NORMALITY REJECTED	1.566918	0.506	0.577	0.692	0.807	0.960
5.00	5	MAY BE NORMAL	0.841590	0.506	0.577	0.692	0.807	0.960
5.00	50	NORMALITY REJECTED	0.998077	0.506	0.577	0.692	0.807	0.960

5.00	100	NORMALITY REJECTED	2.040994	0.506	0.577	0.692	0.807	0.960
5.00	500	NORMALITY REJECTED	1.363133	0.506	0.577	0.692	0.807	0.960
50.00	5	MAY BE NORMAL	0.634215	0.506	0.577	0.692	0.807	0.960
50.00	50	MAY BE NORMAL	0.225983	0.506	0.577	0.692	0.807	0.960
50.00	100	MAY BE NORMAL	0.186687	0.506	0.577	0.692	0.807	0.960
50.00	500	NORMALITY REJECTED	3.158217	0.506	0.577	0.692	0.807	0.960

Metric: Best candidate throughput (races/s) normalized to the fastest candidate.

C	W	NORMAL?	A2	15%	10%	5%	2.5%	1%
0.01	5	NORMALITY REJECTED	0.975432	0.506	0.577	0.692	0.807	0.960
0.01	50	NORMALITY REJECTED	1.368368	0.506	0.577	0.692	0.807	0.960
0.01	100	NORMALITY REJECTED	1.221460	0.506	0.577	0.692	0.807	0.960
0.01	500	NORMALITY REJECTED	2.325712	0.506	0.577	0.692	0.807	0.960
0.10	5	MAY BE NORMAL	0.537387	0.506	0.577	0.692	0.807	0.960
0.10	50	NORMALITY REJECTED	2.700307	0.506	0.577	0.692	0.807	0.960
0.10	100	NORMALITY REJECTED	1.449773	0.506	0.577	0.692	0.807	0.960
0.10	500	NORMALITY REJECTED	2.933374	0.506	0.577	0.692	0.807	0.960
0.50	5	NORMALITY REJECTED	1.398640	0.506	0.577	0.692	0.807	0.960
0.50	50	NORMALITY REJECTED	1.053361	0.506	0.577	0.692	0.807	0.960
0.50	100	NORMALITY REJECTED	2.548517	0.506	0.577	0.692	0.807	0.960
0.50	500	NORMALITY REJECTED	2.063891	0.506	0.577	0.692	0.807	0.960
5.00	5	MAY BE NORMAL	0.857102	0.506	0.577	0.692	0.807	0.960
5.00	50	NORMALITY REJECTED	0.968675	0.506	0.577	0.692	0.807	0.960
5.00	100	NORMALITY REJECTED	1.719031	0.506	0.577	0.692	0.807	0.960
5.00	500	NORMALITY REJECTED	0.965637	0.506	0.577	0.692	0.807	0.960
50.00	5	MAY BE NORMAL	0.379293	0.506	0.577	0.692	0.807	0.960
50.00	50	MAY BE NORMAL	0.457801	0.506	0.577	0.692	0.807	0.960
50.00	100	MAY BE NORMAL	0.274088	0.506	0.577	0.692	0.807	0.960
50.00	500	MAY BE NORMAL	0.438109	0.506	0.577	0.692	0.807	0.960

Metric: Time to convergence (number of races) normalized to the lowest one achieved.

C	W	NORMAL?	A2	15%	10%	5%	2.5%	1%
0.01	5	NORMALITY REJECTED	3.554670	0.506	0.577	0.692	0.807	0.960
0.01	50	NORMALITY REJECTED	3.545934	0.506	0.577	0.692	0.807	0.960
0.01	100	NORMALITY REJECTED	2.556568	0.506	0.577	0.692	0.807	0.960
0.01	500	NORMALITY REJECTED	3.782695	0.506	0.577	0.692	0.807	0.960
0.10	5	NORMALITY REJECTED	4.239544	0.506	0.577	0.692	0.807	0.960
0.10	50	NORMALITY REJECTED	5.069045	0.506	0.577	0.692	0.807	0.960
0.10	100	NORMALITY REJECTED	1.891674	0.506	0.577	0.692	0.807	0.960
0.10	500	NORMALITY REJECTED	3.434625	0.506	0.577	0.692	0.807	0.960
0.50	5	NORMALITY REJECTED	3.571500	0.506	0.577	0.692	0.807	0.960
0.50	50	NORMALITY REJECTED	2.864744	0.506	0.577	0.692	0.807	0.960
0.50	100	NORMALITY REJECTED	3.557405	0.506	0.577	0.692	0.807	0.960

0.50	500	NORMALITY REJECTED	3.395737	0.506	0.577	0.692	0.807	0.960
5.00	5	NORMALITY REJECTED	1.195812	0.506	0.577	0.692	0.807	0.960
5.00	50	NORMALITY REJECTED	3.813249	0.506	0.577	0.692	0.807	0.960
5.00	100	NORMALITY REJECTED	2.618921	0.506	0.577	0.692	0.807	0.960
5.00	500	NORMALITY REJECTED	2.580556	0.506	0.577	0.692	0.807	0.960
50.00	5	MAY BE NORMAL	0.507667	0.506	0.577	0.692	0.807	0.960
50.00	50	NORMALITY REJECTED	1.777829	0.506	0.577	0.692	0.807	0.960
50.00	100	NORMALITY REJECTED	1.498423	0.506	0.577	0.692	0.807	0.960
50.00	500	NORMALITY REJECTED	0.985673	0.506	0.577	0.692	0.807	0.960

Matrix Approximation

Metric: Mean throughput (races/s) normalized to the fastest configuration.

C	W	NORMAL?	A2	15%	10%	5%	2.5%	1%
0.01	5	NORMALITY REJECTED	6.471845	0.506	0.577	0.692	0.807	0.960
0.01	50	MAY BE NORMAL	0.787854	0.506	0.577	0.692	0.807	0.960
0.01	100	NORMALITY REJECTED	6.278897	0.506	0.577	0.692	0.807	0.960
0.01	500	NORMALITY REJECTED	6.195502	0.506	0.577	0.692	0.807	0.960
0.10	5	NORMALITY REJECTED	2.312002	0.506	0.577	0.692	0.807	0.960
0.10	50	NORMALITY REJECTED	4.900069	0.506	0.577	0.692	0.807	0.960
0.10	100	NORMALITY REJECTED	2.408656	0.506	0.577	0.692	0.807	0.960
0.10	500	NORMALITY REJECTED	4.916105	0.506	0.577	0.692	0.807	0.960
0.50	5	NORMALITY REJECTED	5.163870	0.506	0.577	0.692	0.807	0.960
0.50	50	NORMALITY REJECTED	2.858056	0.506	0.577	0.692	0.807	0.960
0.50	100	NORMALITY REJECTED	1.490699	0.506	0.577	0.692	0.807	0.960
0.50	500	NORMALITY REJECTED	5.784569	0.506	0.577	0.692	0.807	0.960
5.00	5	NORMALITY REJECTED	3.425115	0.506	0.577	0.692	0.807	0.960
5.00	50	NORMALITY REJECTED	6.456761	0.506	0.577	0.692	0.807	0.960
5.00	100	NORMALITY REJECTED	5.934514	0.506	0.577	0.692	0.807	0.960
5.00	500	NORMALITY REJECTED	6.104563	0.506	0.577	0.692	0.807	0.960
50.00	5	NORMALITY REJECTED	3.471913	0.506	0.577	0.692	0.807	0.960
50.00	50	NORMALITY REJECTED	1.962627	0.506	0.577	0.692	0.807	0.960
50.00	100	NORMALITY REJECTED	5.913393	0.506	0.577	0.692	0.807	0.960
50.00	500	NORMALITY REJECTED	6.085765	0.506	0.577	0.692	0.807	0.960

Metric: Best candidate throughput (races/s) normalized to the fastest candidate.

C	W	NORMAL?	A2	15%	10%	5%	2.5%	1%
0.01	5	NORMALITY REJECTED	6.914236	0.506	0.577	0.692	0.807	0.960
0.01	50	NORMALITY REJECTED	1.972320	0.506	0.577	0.692	0.807	0.960
0.01	100	NORMALITY REJECTED	6.948895	0.506	0.577	0.692	0.807	0.960
0.01	500	NORMALITY REJECTED	6.793023	0.506	0.577	0.692	0.807	0.960
0.10	5	NORMALITY REJECTED	1.386557	0.506	0.577	0.692	0.807	0.960

0.10	50	NORMALITY REJECTED	6.816205	0.506	0.577	0.692	0.807	0.960
0.10	100	NORMALITY REJECTED	1.003870	0.506	0.577	0.692	0.807	0.960
0.10	500	NORMALITY REJECTED	5.740119	0.506	0.577	0.692	0.807	0.960
0.50	5	NORMALITY REJECTED	2.993490	0.506	0.577	0.692	0.807	0.960
0.50	50	MAY BE NORMAL	0.269584	0.506	0.577	0.692	0.807	0.960
0.50	100	MAY BE NORMAL	0.491067	0.506	0.577	0.692	0.807	0.960
0.50	500	NORMALITY REJECTED	6.853773	0.506	0.577	0.692	0.807	0.960
5.00	5	NORMALITY REJECTED	1.001114	0.506	0.577	0.692	0.807	0.960
5.00	50	NORMALITY REJECTED	7.022443	0.506	0.577	0.692	0.807	0.960
5.00	100	NORMALITY REJECTED	6.400443	0.506	0.577	0.692	0.807	0.960
5.00	500	NORMALITY REJECTED	6.329968	0.506	0.577	0.692	0.807	0.960
50.00	5	MAY BE NORMAL	0.717650	0.506	0.577	0.692	0.807	0.960
50.00	50	MAY BE NORMAL	0.356491	0.506	0.577	0.692	0.807	0.960
50.00	100	NORMALITY REJECTED	6.360595	0.506	0.577	0.692	0.807	0.960
50.00	500	NORMALITY REJECTED	6.969796	0.506	0.577	0.692	0.807	0.960

Metric: Accuracy Root Mean Square Error (RMSE) normalized to the lowest one achieved.

C	W	NORMAL?	A2	15%	10%	5%	2.5%	1%
0.01	5	NORMALITY REJECTED	4.828264	0.506	0.577	0.692	0.807	0.960
0.01	50	NORMALITY REJECTED	3.077410	0.506	0.577	0.692	0.807	0.960
0.01	100	NORMALITY REJECTED	3.891065	0.506	0.577	0.692	0.807	0.960
0.01	500	NORMALITY REJECTED	3.418851	0.506	0.577	0.692	0.807	0.960
0.10	5	NORMALITY REJECTED	4.227304	0.506	0.577	0.692	0.807	0.960
0.10	50	NORMALITY REJECTED	4.768180	0.506	0.577	0.692	0.807	0.960
0.10	100	NORMALITY REJECTED	4.503351	0.506	0.577	0.692	0.807	0.960
0.10	500	NORMALITY REJECTED	5.061175	0.506	0.577	0.692	0.807	0.960
0.50	5	NORMALITY REJECTED	6.611483	0.506	0.577	0.692	0.807	0.960
0.50	50	NORMALITY REJECTED	4.128878	0.506	0.577	0.692	0.807	0.960
0.50	100	NORMALITY REJECTED	4.355959	0.506	0.577	0.692	0.807	0.960
0.50	500	NORMALITY REJECTED	4.324321	0.506	0.577	0.692	0.807	0.960
5.00	5	NORMALITY REJECTED	3.527708	0.506	0.577	0.692	0.807	0.960
5.00	50	NORMALITY REJECTED	2.508644	0.506	0.577	0.692	0.807	0.960
5.00	100	NORMALITY REJECTED	3.605618	0.506	0.577	0.692	0.807	0.960
5.00	500	NORMALITY REJECTED	3.444852	0.506	0.577	0.692	0.807	0.960
50.00	5	NORMALITY REJECTED	4.595070	0.506	0.577	0.692	0.807	0.960
50.00	50	NORMALITY REJECTED	3.563231	0.506	0.577	0.692	0.807	0.960
50.00	100	NORMALITY REJECTED	1.017475	0.506	0.577	0.692	0.807	0.960
50.00	500	NORMALITY REJECTED	3.654126	0.506	0.577	0.692	0.807	0.960

Metric: Percentage of races that met accuracy target normalized to the highest one achieved.

C	W	NORMAL?	A2	15%	10%	5%	2.5%	1%
0.01	5	NORMALITY REJECTED	inf	0.506	0.577	0.692	0.807	0.960
0.01	50	NORMALITY REJECTED	inf	0.506	0.577	0.692	0.807	0.960

0.01	100	NORMALITY REJECTED	inf	0.506	0.577	0.692	0.807	0.960
0.01	500	NORMALITY REJECTED	inf	0.506	0.577	0.692	0.807	0.960
0.10	5	NORMALITY REJECTED	inf	0.506	0.577	0.692	0.807	0.960
0.10	50	NORMALITY REJECTED	inf	0.506	0.577	0.692	0.807	0.960
0.10	100	NORMALITY REJECTED	inf	0.506	0.577	0.692	0.807	0.960
0.10	500	NORMALITY REJECTED	inf	0.506	0.577	0.692	0.807	0.960
0.50	5	NORMALITY REJECTED	inf	0.506	0.577	0.692	0.807	0.960
0.50	50	NORMALITY REJECTED	inf	0.506	0.577	0.692	0.807	0.960
0.50	100	NORMALITY REJECTED	inf	0.506	0.577	0.692	0.807	0.960
0.50	500	NORMALITY REJECTED	inf	0.506	0.577	0.692	0.807	0.960
5.00	5	NORMALITY REJECTED	inf	0.506	0.577	0.692	0.807	0.960
5.00	50	NORMALITY REJECTED	inf	0.506	0.577	0.692	0.807	0.960
5.00	100	NORMALITY REJECTED	inf	0.506	0.577	0.692	0.807	0.960
5.00	500	NORMALITY REJECTED	inf	0.506	0.577	0.692	0.807	0.960
50.00	5	NORMALITY REJECTED	inf	0.506	0.577	0.692	0.807	0.960
50.00	50	NORMALITY REJECTED	inf	0.506	0.577	0.692	0.807	0.960
50.00	100	NORMALITY REJECTED	inf	0.506	0.577	0.692	0.807	0.960
50.00	500	NORMALITY REJECTED	inf	0.506	0.577	0.692	0.807	0.960

Metric: Time to convergence (number of races) normalized to the lowest one achieved.

C	W	NORMAL?	A2	15%	10%	5%	2.5%	1%
0.01	5	NORMALITY REJECTED	2.767464	0.506	0.577	0.692	0.807	0.960
0.01	50	NORMALITY REJECTED	3.110145	0.506	0.577	0.692	0.807	0.960
0.01	100	NORMALITY REJECTED	3.396807	0.506	0.577	0.692	0.807	0.960
0.01	500	NORMALITY REJECTED	4.005008	0.506	0.577	0.692	0.807	0.960
0.10	5	NORMALITY REJECTED	3.376279	0.506	0.577	0.692	0.807	0.960
0.10	50	NORMALITY REJECTED	1.888039	0.506	0.577	0.692	0.807	0.960
0.10	100	NORMALITY REJECTED	4.212349	0.506	0.577	0.692	0.807	0.960
0.10	500	NORMALITY REJECTED	1.585284	0.506	0.577	0.692	0.807	0.960
0.50	5	NORMALITY REJECTED	3.507387	0.506	0.577	0.692	0.807	0.960
0.50	50	NORMALITY REJECTED	3.969521	0.506	0.577	0.692	0.807	0.960
0.50	100	NORMALITY REJECTED	3.507387	0.506	0.577	0.692	0.807	0.960
0.50	500	NORMALITY REJECTED	4.020303	0.506	0.577	0.692	0.807	0.960
5.00	5	NORMALITY REJECTED	3.654465	0.506	0.577	0.692	0.807	0.960
5.00	50	NORMALITY REJECTED	3.691324	0.506	0.577	0.692	0.807	0.960
5.00	100	NORMALITY REJECTED	1.842899	0.506	0.577	0.692	0.807	0.960
5.00	500	NORMALITY REJECTED	1.944847	0.506	0.577	0.692	0.807	0.960
50.00	5	NORMALITY REJECTED	4.005008	0.506	0.577	0.692	0.807	0.960
50.00	50	NORMALITY REJECTED	4.608751	0.506	0.577	0.692	0.807	0.960
50.00	100	NORMALITY REJECTED	2.007202	0.506	0.577	0.692	0.807	0.960
50.00	500	NORMALITY REJECTED	3.335577	0.506	0.577	0.692	0.807	0.960

Poisson

Metric: Mean throughput (races/s) normalized to the fastest configuration.

C	W	NORMAL?	A2	15%	10%	5%	2.5%	1%
0.01	5	NORMALITY REJECTED	1.792714	0.506	0.577	0.692	0.807	0.960
0.01	50	NORMALITY REJECTED	1.722617	0.506	0.577	0.692	0.807	0.960
0.01	100	NORMALITY REJECTED	4.956789	0.506	0.577	0.692	0.807	0.960
0.01	500	NORMALITY REJECTED	1.913765	0.506	0.577	0.692	0.807	0.960
0.10	5	NORMALITY REJECTED	1.252561	0.506	0.577	0.692	0.807	0.960
0.10	50	NORMALITY REJECTED	5.856768	0.506	0.577	0.692	0.807	0.960
0.10	100	NORMALITY REJECTED	3.280718	0.506	0.577	0.692	0.807	0.960
0.10	500	NORMALITY REJECTED	3.099851	0.506	0.577	0.692	0.807	0.960
0.50	5	NORMALITY REJECTED	1.678140	0.506	0.577	0.692	0.807	0.960
0.50	50	NORMALITY REJECTED	5.005080	0.506	0.577	0.692	0.807	0.960
0.50	100	NORMALITY REJECTED	3.639688	0.506	0.577	0.692	0.807	0.960
0.50	500	NORMALITY REJECTED	5.981053	0.506	0.577	0.692	0.807	0.960
5.00	5	NORMALITY REJECTED	3.266817	0.506	0.577	0.692	0.807	0.960
5.00	50	NORMALITY REJECTED	6.229375	0.506	0.577	0.692	0.807	0.960
5.00	100	NORMALITY REJECTED	4.698265	0.506	0.577	0.692	0.807	0.960
5.00	500	NORMALITY REJECTED	4.625570	0.506	0.577	0.692	0.807	0.960
50.00	5	MAY BE NORMAL	0.741342	0.506	0.577	0.692	0.807	0.960
50.00	50	NORMALITY REJECTED	1.563511	0.506	0.577	0.692	0.807	0.960
50.00	100	NORMALITY REJECTED	1.693622	0.506	0.577	0.692	0.807	0.960
50.00	500	NORMALITY REJECTED	1.786646	0.506	0.577	0.692	0.807	0.960

Metric: Best candidate throughput (races/s) normalized to the fastest candidate.

C	W	NORMAL?	A2	15%	10%	5%	2.5%	1%
0.01	5	NORMALITY REJECTED	1.135881	0.506	0.577	0.692	0.807	0.960
0.01	50	NORMALITY REJECTED	6.716379	0.506	0.577	0.692	0.807	0.960
0.01	100	MAY BE NORMAL	0.442883	0.506	0.577	0.692	0.807	0.960
0.01	500	NORMALITY REJECTED	6.800512	0.506	0.577	0.692	0.807	0.960
0.10	5	MAY BE NORMAL	0.638555	0.506	0.577	0.692	0.807	0.960
0.10	50	MAY BE NORMAL	0.758763	0.506	0.577	0.692	0.807	0.960
0.10	100	MAY BE NORMAL	0.388328	0.506	0.577	0.692	0.807	0.960
0.10	500	NORMALITY REJECTED	6.156075	0.506	0.577	0.692	0.807	0.960
0.50	5	NORMALITY REJECTED	1.040755	0.506	0.577	0.692	0.807	0.960
0.50	50	MAY BE NORMAL	0.390502	0.506	0.577	0.692	0.807	0.960
0.50	100	NORMALITY REJECTED	6.401159	0.506	0.577	0.692	0.807	0.960
0.50	500	MAY BE NORMAL	0.904336	0.506	0.577	0.692	0.807	0.960
5.00	5	MAY BE NORMAL	0.457849	0.506	0.577	0.692	0.807	0.960
5.00	50	MAY BE NORMAL	0.608555	0.506	0.577	0.692	0.807	0.960
5.00	100	MAY BE NORMAL	0.466239	0.506	0.577	0.692	0.807	0.960
5.00	500	NORMALITY REJECTED	4.163870	0.506	0.577	0.692	0.807	0.960
50.00	5	MAY BE NORMAL	0.568534	0.506	0.577	0.692	0.807	0.960
50.00	50	NORMALITY REJECTED	5.523935	0.506	0.577	0.692	0.807	0.960

50.00	100	MAY BE NORMAL	0.560662	0.506	0.577	0.692	0.807	0.960
50.00	500	NORMALITY REJECTED	2.901978	0.506	0.577	0.692	0.807	0.960

Metric: Accuracy Root Mean Square Error (RMSE) normalized to the lowest one achieved.

C	W	NORMAL?	A2	15%	10%	5%	2.5%	1%
0.01	5	NORMALITY REJECTED	5.186767	0.506	0.577	0.692	0.807	0.960
0.01	50	NORMALITY REJECTED	7.000291	0.506	0.577	0.692	0.807	0.960
0.01	100	NORMALITY REJECTED	5.106165	0.506	0.577	0.692	0.807	0.960
0.01	500	NORMALITY REJECTED	4.934784	0.506	0.577	0.692	0.807	0.960
0.10	5	NORMALITY REJECTED	7.176183	0.506	0.577	0.692	0.807	0.960
0.10	50	NORMALITY REJECTED	5.841794	0.506	0.577	0.692	0.807	0.960
0.10	100	NORMALITY REJECTED	3.436117	0.506	0.577	0.692	0.807	0.960
0.10	500	NORMALITY REJECTED	4.393460	0.506	0.577	0.692	0.807	0.960
0.50	5	NORMALITY REJECTED	5.181042	0.506	0.577	0.692	0.807	0.960
0.50	50	NORMALITY REJECTED	5.198230	0.506	0.577	0.692	0.807	0.960
0.50	100	NORMALITY REJECTED	7.009655	0.506	0.577	0.692	0.807	0.960
0.50	500	NORMALITY REJECTED	5.818682	0.506	0.577	0.692	0.807	0.960
5.00	5	NORMALITY REJECTED	5.331403	0.506	0.577	0.692	0.807	0.960
5.00	50	NORMALITY REJECTED	6.314007	0.506	0.577	0.692	0.807	0.960
5.00	100	NORMALITY REJECTED	6.684910	0.506	0.577	0.692	0.807	0.960
5.00	500	NORMALITY REJECTED	3.566855	0.506	0.577	0.692	0.807	0.960
50.00	5	MAY BE NORMAL	0.934914	0.506	0.577	0.692	0.807	0.960
50.00	50	NORMALITY REJECTED	5.436100	0.506	0.577	0.692	0.807	0.960
50.00	100	NORMALITY REJECTED	2.081637	0.506	0.577	0.692	0.807	0.960
50.00	500	NORMALITY REJECTED	3.278293	0.506	0.577	0.692	0.807	0.960

Metric: Percentage of races that met accuracy target normalized to the highest one achieved.

C	W	NORMAL?	A2	15%	10%	5%	2.5%	1%
0.01	5	NORMALITY REJECTED	5.186768	0.506	0.577	0.692	0.807	0.960
0.01	50	NORMALITY REJECTED	2.059319	0.506	0.577	0.692	0.807	0.960
0.01	100	NORMALITY REJECTED	5.045847	0.506	0.577	0.692	0.807	0.960
0.01	500	NORMALITY REJECTED	2.183706	0.506	0.577	0.692	0.807	0.960
0.10	5	NORMALITY REJECTED	7.176183	0.506	0.577	0.692	0.807	0.960
0.10	50	NORMALITY REJECTED	5.786595	0.506	0.577	0.692	0.807	0.960
0.10	100	NORMALITY REJECTED	3.435604	0.506	0.577	0.692	0.807	0.960
0.10	500	NORMALITY REJECTED	3.435375	0.506	0.577	0.692	0.807	0.960
0.50	5	NORMALITY REJECTED	5.181044	0.506	0.577	0.692	0.807	0.960
0.50	50	NORMALITY REJECTED	5.195923	0.506	0.577	0.692	0.807	0.960
0.50	100	NORMALITY REJECTED	3.946034	0.506	0.577	0.692	0.807	0.960
0.50	500	NORMALITY REJECTED	5.814134	0.506	0.577	0.692	0.807	0.960
5.00	5	NORMALITY REJECTED	5.331465	0.506	0.577	0.692	0.807	0.960
5.00	50	NORMALITY REJECTED	6.310959	0.506	0.577	0.692	0.807	0.960
5.00	100	NORMALITY REJECTED	6.669256	0.506	0.577	0.692	0.807	0.960

5.00	500	NORMALITY REJECTED	4.080386	0.506	0.577	0.692	0.807	0.960
50.00	5	MAY BE NORMAL	0.938337	0.506	0.577	0.692	0.807	0.960
50.00	50	NORMALITY REJECTED	2.458384	0.506	0.577	0.692	0.807	0.960
50.00	100	NORMALITY REJECTED	2.062167	0.506	0.577	0.692	0.807	0.960
50.00	500	MAY BE NORMAL	0.893809	0.506	0.577	0.692	0.807	0.960

Metric: Time to convergence (number of races) normalized to the lowest one achieved.

C	W	NORMAL?	A2	15%	10%	5%	2.5%	1%
0.01	5	NORMALITY REJECTED	2.487122	0.506	0.577	0.692	0.807	0.960
0.01	50	NORMALITY REJECTED	3.788360	0.506	0.577	0.692	0.807	0.960
0.01	100	NORMALITY REJECTED	5.187950	0.506	0.577	0.692	0.807	0.960
0.01	500	NORMALITY REJECTED	4.333114	0.506	0.577	0.692	0.807	0.960
0.10	5	NORMALITY REJECTED	4.071960	0.506	0.577	0.692	0.807	0.960
0.10	50	NORMALITY REJECTED	4.423899	0.506	0.577	0.692	0.807	0.960
0.10	100	NORMALITY REJECTED	4.758829	0.506	0.577	0.692	0.807	0.960
0.10	500	NORMALITY REJECTED	4.243521	0.506	0.577	0.692	0.807	0.960
0.50	5	NORMALITY REJECTED	3.940009	0.506	0.577	0.692	0.807	0.960
0.50	50	NORMALITY REJECTED	3.543860	0.506	0.577	0.692	0.807	0.960
0.50	100	NORMALITY REJECTED	5.493728	0.506	0.577	0.692	0.807	0.960
0.50	500	NORMALITY REJECTED	2.498127	0.506	0.577	0.692	0.807	0.960
5.00	5	NORMALITY REJECTED	3.830711	0.506	0.577	0.692	0.807	0.960
5.00	50	NORMALITY REJECTED	2.616884	0.506	0.577	0.692	0.807	0.960
5.00	100	NORMALITY REJECTED	4.075290	0.506	0.577	0.692	0.807	0.960
5.00	500	NORMALITY REJECTED	3.988676	0.506	0.577	0.692	0.807	0.960
50.00	5	NORMALITY REJECTED	5.142796	0.506	0.577	0.692	0.807	0.960
50.00	50	NORMALITY REJECTED	4.108465	0.506	0.577	0.692	0.807	0.960
50.00	100	NORMALITY REJECTED	3.165549	0.506	0.577	0.692	0.807	0.960
50.00	500	NORMALITY REJECTED	1.880216	0.506	0.577	0.692	0.807	0.960

Binpacking

Metric: Mean throughput (races/s) normalized to the fastest configuration.

C	W	NORMAL?	A2	15%	10%	5%	2.5%	1%
0.01	5	MAY BE NORMAL	0.515913	0.506	0.577	0.692	0.807	0.960
0.01	50	MAY BE NORMAL	0.667907	0.506	0.577	0.692	0.807	0.960
0.01	100	MAY BE NORMAL	0.345994	0.506	0.577	0.692	0.807	0.960
0.01	500	MAY BE NORMAL	0.430656	0.506	0.577	0.692	0.807	0.960
0.10	5	MAY BE NORMAL	0.733656	0.506	0.577	0.692	0.807	0.960
0.10	50	NORMALITY REJECTED	1.392133	0.506	0.577	0.692	0.807	0.960
0.10	100	NORMALITY REJECTED	1.545454	0.506	0.577	0.692	0.807	0.960
0.10	500	MAY BE NORMAL	0.341793	0.506	0.577	0.692	0.807	0.960
0.50	5	MAY BE NORMAL	0.490482	0.506	0.577	0.692	0.807	0.960

0.50	50	NORMALITY REJECTED	1.898467	0.506	0.577	0.692	0.807	0.960
0.50	100	MAY BE NORMAL	0.856558	0.506	0.577	0.692	0.807	0.960
0.50	500	MAY BE NORMAL	0.674900	0.506	0.577	0.692	0.807	0.960
5.00	5	MAY BE NORMAL	0.192859	0.506	0.577	0.692	0.807	0.960
5.00	50	NORMALITY REJECTED	1.166367	0.506	0.577	0.692	0.807	0.960
5.00	100	MAY BE NORMAL	0.418981	0.506	0.577	0.692	0.807	0.960
5.00	500	MAY BE NORMAL	0.591486	0.506	0.577	0.692	0.807	0.960
50.00	5	MAY BE NORMAL	0.907837	0.506	0.577	0.692	0.807	0.960
50.00	50	MAY BE NORMAL	0.290021	0.506	0.577	0.692	0.807	0.960
50.00	100	MAY BE NORMAL	0.467232	0.506	0.577	0.692	0.807	0.960
50.00	500	MAY BE NORMAL	0.326396	0.506	0.577	0.692	0.807	0.960

Metric: Best candidate throughput (races/s) normalized to the fastest candidate.

C	W	NORMAL?	A2	15%	10%	5%	2.5%	1%
0.01	5	NORMALITY REJECTED	1.205326	0.543	0.618	0.742	0.865	1.029
0.01	50	NORMALITY REJECTED	2.931485	0.506	0.577	0.692	0.807	0.960
0.01	100	NORMALITY REJECTED	5.550726	0.505	0.575	0.690	0.804	0.957
0.01	500	NORMALITY REJECTED	1.721769	0.505	0.575	0.690	0.804	0.957
0.10	5	MAY BE NORMAL	0.377386	0.720	0.820	0.984	1.148	1.365
0.10	50	NORMALITY REJECTED	4.604009	0.506	0.577	0.692	0.807	0.960
0.10	100	NORMALITY REJECTED	5.275814	0.505	0.575	0.690	0.804	0.957
0.10	500	NORMALITY REJECTED	2.783210	0.505	0.575	0.690	0.804	0.957
0.50	5	MAY BE NORMAL	1.069728	0.720	0.820	0.984	1.148	1.365
0.50	50	NORMALITY REJECTED	3.664560	0.505	0.575	0.690	0.804	0.957
0.50	100	NORMALITY REJECTED	5.209623	0.505	0.575	0.690	0.804	0.957
0.50	500	NORMALITY REJECTED	2.861428	0.505	0.575	0.690	0.804	0.957
5.00	5	NORMALITY REJECTED	2.938824	0.497	0.566	0.679	0.792	0.942
5.00	50	NORMALITY REJECTED	4.452127	0.505	0.575	0.690	0.804	0.957
5.00	100	NORMALITY REJECTED	1.624272	0.506	0.577	0.692	0.807	0.960
5.00	500	NORMALITY REJECTED	2.649097	0.505	0.575	0.690	0.804	0.957
50.00	5	NORMALITY REJECTED	3.607776	0.506	0.577	0.692	0.807	0.960
50.00	50	MAY BE NORMAL	0.616544	0.506	0.577	0.692	0.807	0.960
50.00	100	NORMALITY REJECTED	2.445417	0.506	0.577	0.692	0.807	0.960
50.00	500	MAY BE NORMAL	0.506259	0.506	0.577	0.692	0.807	0.960

Metric: Accuracy Root Mean Square Error (RMSE) normalized to the lowest one achieved.

C	W	NORMAL?	A2	15%	10%	5%	2.5%	1%
0.01	5	MAY BE NORMAL	0.925886	0.506	0.577	0.692	0.807	0.960
0.01	50	NORMALITY REJECTED	1.124714	0.506	0.577	0.692	0.807	0.960
0.01	100	NORMALITY REJECTED	2.108852	0.506	0.577	0.692	0.807	0.960
0.01	500	NORMALITY REJECTED	0.980711	0.506	0.577	0.692	0.807	0.960
0.10	5	NORMALITY REJECTED	1.005198	0.506	0.577	0.692	0.807	0.960
0.10	50	NORMALITY REJECTED	2.324642	0.506	0.577	0.692	0.807	0.960

0.10	100	MAY BE NORMAL	0.484641	0.506	0.577	0.692	0.807	0.960
0.10	500	NORMALITY REJECTED	1.203023	0.506	0.577	0.692	0.807	0.960
0.50	5	NORMALITY REJECTED	1.395352	0.506	0.577	0.692	0.807	0.960
0.50	50	NORMALITY REJECTED	1.301472	0.506	0.577	0.692	0.807	0.960
0.50	100	NORMALITY REJECTED	2.121076	0.506	0.577	0.692	0.807	0.960
0.50	500	MAY BE NORMAL	0.907868	0.506	0.577	0.692	0.807	0.960
5.00	5	MAY BE NORMAL	0.717962	0.506	0.577	0.692	0.807	0.960
5.00	50	NORMALITY REJECTED	1.075917	0.506	0.577	0.692	0.807	0.960
5.00	100	MAY BE NORMAL	0.883520	0.506	0.577	0.692	0.807	0.960
5.00	500	MAY BE NORMAL	0.957092	0.506	0.577	0.692	0.807	0.960
50.00	5	MAY BE NORMAL	0.352204	0.506	0.577	0.692	0.807	0.960
50.00	50	MAY BE NORMAL	0.617013	0.506	0.577	0.692	0.807	0.960
50.00	100	MAY BE NORMAL	0.242287	0.506	0.577	0.692	0.807	0.960
50.00	500	MAY BE NORMAL	0.217710	0.506	0.577	0.692	0.807	0.960

Metric: Percentage of races that met accuracy target normalized to the highest one achieved.

C	W	NORMAL?	A2	15%	10%	5%	2.5%	1%
0.01	5	NORMALITY REJECTED	3.047013	0.506	0.577	0.692	0.807	0.960
0.01	50	NORMALITY REJECTED	1.065901	0.506	0.577	0.692	0.807	0.960
0.01	100	NORMALITY REJECTED	1.152731	0.506	0.577	0.692	0.807	0.960
0.01	500	MAY BE NORMAL	0.569835	0.506	0.577	0.692	0.807	0.960
0.10	5	NORMALITY REJECTED	4.444035	0.506	0.577	0.692	0.807	0.960
0.10	50	MAY BE NORMAL	0.543990	0.506	0.577	0.692	0.807	0.960
0.10	100	MAY BE NORMAL	0.461665	0.506	0.577	0.692	0.807	0.960
0.10	500	NORMALITY REJECTED	1.051307	0.506	0.577	0.692	0.807	0.960
0.50	5	NORMALITY REJECTED	4.254145	0.506	0.577	0.692	0.807	0.960
0.50	50	MAY BE NORMAL	0.546880	0.506	0.577	0.692	0.807	0.960
0.50	100	MAY BE NORMAL	0.932822	0.506	0.577	0.692	0.807	0.960
0.50	500	MAY BE NORMAL	0.405789	0.506	0.577	0.692	0.807	0.960
5.00	5	NORMALITY REJECTED	2.422840	0.506	0.577	0.692	0.807	0.960
5.00	50	MAY BE NORMAL	0.753084	0.506	0.577	0.692	0.807	0.960
5.00	100	MAY BE NORMAL	0.356659	0.506	0.577	0.692	0.807	0.960
5.00	500	MAY BE NORMAL	0.432178	0.506	0.577	0.692	0.807	0.960
50.00	5	MAY BE NORMAL	0.535021	0.506	0.577	0.692	0.807	0.960
50.00	50	MAY BE NORMAL	0.853794	0.506	0.577	0.692	0.807	0.960
50.00	100	MAY BE NORMAL	0.525790	0.506	0.577	0.692	0.807	0.960
50.00	500	MAY BE NORMAL	0.310162	0.506	0.577	0.692	0.807	0.960

Metric: Time to convergence (number of races) normalized to the lowest one achieved.

C	W	NORMAL?	A2	15%	10%	5%	2.5%	1%
0.01	5	MAY BE NORMAL	0.327279	0.543	0.618	0.742	0.865	1.029
0.01	50	NORMALITY REJECTED	1.347578	0.506	0.577	0.692	0.807	0.960
0.01	100	NORMALITY REJECTED	2.505134	0.505	0.575	0.690	0.804	0.957

0.01	500	NORMALITY REJECTED	3.774588	0.505	0.575	0.690	0.804	0.957
0.10	5	MAY BE NORMAL	0.459079	0.720	0.820	0.984	1.148	1.365
0.10	50	NORMALITY REJECTED	3.321063	0.506	0.577	0.692	0.807	0.960
0.10	100	NORMALITY REJECTED	2.585089	0.505	0.575	0.690	0.804	0.957
0.10	500	NORMALITY REJECTED	2.749053	0.505	0.575	0.690	0.804	0.957
0.50	5	MAY BE NORMAL	0.459141	0.720	0.820	0.984	1.148	1.365
0.50	50	NORMALITY REJECTED	4.102456	0.505	0.575	0.690	0.804	0.957
0.50	100	NORMALITY REJECTED	1.626919	0.505	0.575	0.690	0.804	0.957
0.50	500	NORMALITY REJECTED	2.091247	0.505	0.575	0.690	0.804	0.957
5.00	5	MAY BE NORMAL	0.153680	0.497	0.566	0.679	0.792	0.942
5.00	50	NORMALITY REJECTED	2.907449	0.505	0.575	0.690	0.804	0.957
5.00	100	MAY BE NORMAL	0.449190	0.506	0.577	0.692	0.807	0.960
5.00	500	MAY BE NORMAL	0.278148	0.505	0.575	0.690	0.804	0.957
50.00	5	NORMALITY REJECTED	1.100984	0.506	0.577	0.692	0.807	0.960
50.00	50	NORMALITY REJECTED	1.970365	0.506	0.577	0.692	0.807	0.960
50.00	100	NORMALITY REJECTED	1.002109	0.506	0.577	0.692	0.807	0.960
50.00	500	NORMALITY REJECTED	1.003541	0.506	0.577	0.692	0.807	0.960

A.2 Hyperparameter Runs: Significance Testing

All significance tests were performed using the Wilcoxon signed-rank test from the Python SciPy library included with Ubuntu 11.04 (`scipy.stats.wilcoxon`). `n` is the number of samples (separate runs of the autotuner). `SS?` is `yes` if the configuration was significantly different from the best one with respect to the current metric (p-value < 0.05), or `no` otherwise. The `OPTIMAL?` column marks which configuration was optimal. `z-stat` and `p-val` are the z-statistic and the p-value, respectively. Whenever samples for the given C, W configuration are exactly the same as best, the phrase *equal* appears instead of z and p values.

Note that the Wilcoxon signed-rank test requires the two samples to be of the same length, which is not always true in our setting. For example, some C, W configurations might not always converge, in which case not all runs will provide a "convergence time" sample. Whenever the optimal and current configurations do not have equal-length samples, we trim the extra tail samples from one or the other and invoke the Wilcoxon test on such trimmed samples.

A.2.1 Xeon8

Sort

Metric: Mean throughput (races/s) normalized to the fastest configuration.

C	W	n	SS?	OPTIMAL?	z-stat	p-val
0.01	5	30	yes		1.000000	0.000002
0.01	50	30	yes		8.000000	0.000004
0.01	100	30	yes		11.000000	0.000005
0.01	500	30	yes		4.000000	0.000003
0.10	5	30	yes		1.000000	0.000002
0.10	50	30	yes		3.000000	0.000002
0.10	100	30	yes		7.000000	0.000004
0.10	500	30	yes		7.000000	0.000004
0.50	5	30	yes		4.000000	0.000003
0.50	50	30	yes		11.000000	0.000005
0.50	100	30	yes		25.000000	0.000020
0.50	500	30	yes		13.000000	0.000006
5.00	5	30	yes		6.000000	0.000003
5.00	50	30	yes		65.000000	0.000571
5.00	100	30	yes		87.000000	0.002765
5.00	500	30	yes		53.000000	0.000222
50.00	5	30	yes		45.000000	0.000115
50.00	50	30	no		199.000000	0.490798
50.00	100	30	n/a	*	equal	equal
50.00	500	30	no		193.000000	0.416534

Metric: Best candidate throughput (races/s) normalized to the fastest candidate.

C	W	n	SS?	OPTIMAL?	z-stat	p-val
0.01	5	30	yes		1.000000	0.000002
0.01	50	30	yes		8.000000	0.000004
0.01	100	30	yes		1.000000	0.000002
0.01	500	30	yes		5.000000	0.000003
0.10	5	30	yes		0.000000	0.000002
0.10	50	30	yes		5.000000	0.000003
0.10	100	30	yes		6.000000	0.000003
0.10	500	30	yes		11.000000	0.000005
0.50	5	30	yes		0.000000	0.000002
0.50	50	30	yes		0.000000	0.000002
0.50	100	30	yes		12.000000	0.000006
0.50	500	30	yes		14.000000	0.000007
5.00	5	30	yes		0.000000	0.000002
5.00	50	30	yes		30.000000	0.000031
5.00	100	30	yes		37.000000	0.000058
5.00	500	30	yes		80.000000	0.001709

50.00	5	30	yes		2.000000	0.000002
50.00	50	30	yes		132.000000	0.038723
50.00	100	30	no		151.000000	0.093676
50.00	500	30	n/a	*	equal	equal

Metric: Time to convergence (number of races) normalized to the lowest one achieved.

C	W	n	SS?	OPTIMAL?	z-stat	p-val
0.01	5	30	no		138.000000	0.051931
0.01	50	30	yes		88.000000	0.002957
0.01	100	30	yes		46.000000	0.000126
0.01	500	30	yes		58.000000	0.000332
0.10	5	30	yes		141.500000	0.049494
0.10	50	30	yes		78.000000	0.001484
0.10	100	30	yes		14.000000	0.000010
0.10	500	30	yes		83.000000	0.002105
0.50	5	30	no		188.500000	0.207195
0.50	50	30	yes		104.500000	0.004825
0.50	100	30	yes		87.000000	0.002765
0.50	500	30	yes		20.000000	0.000012
5.00	5	30	n/a	*	equal	equal
5.00	50	30	yes		57.500000	0.000282
5.00	100	30	yes		57.000000	0.000306
5.00	500	30	yes		26.000000	0.000042
50.00	5	30	yes		101.000000	0.011765
50.00	50	30	yes		21.000000	0.000014
50.00	100	30	yes		24.000000	0.000018
50.00	500	30	yes		54.000000	0.000241

Matrix Approximation

Metric: Mean throughput (races/s) normalized to the fastest configuration.

C	W	n	SS?	OPTIMAL?	z-stat	p-val
0.01	5	30	no		161.000000	0.141390
0.01	50	30	no		191.000000	0.393334
0.01	100	30	no		172.000000	0.213358
0.01	500	30	n/a	*	equal	equal
0.10	5	30	no		169.000000	0.191522
0.10	50	30	yes		120.000000	0.020671
0.10	100	30	no		176.000000	0.245190
0.10	500	30	no		162.000000	0.147040
0.50	5	30	no		224.000000	0.861213
0.50	50	30	no		200.000000	0.503833

0.50	100	30	no		173.000000	0.221022
0.50	500	30	yes		135.000000	0.044919
5.00	5	30	no		219.000000	0.781264
5.00	50	30	yes		137.000000	0.049498
5.00	100	30	yes		137.000000	0.049498
5.00	500	30	yes		129.000000	0.033269
50.00	5	30	yes		121.000000	0.021827
50.00	50	30	no		142.000000	0.062683
50.00	100	30	yes		135.000000	0.044919
50.00	500	30	yes		121.000000	0.021827

Metric: Best candidate throughput (races/s) normalized to the fastest candidate.

C	W	n	SS?	OPTIMAL?	z-stat	p-val
0.01	5	30	no		166.000000	0.171376
0.01	50	30	no		150.000000	0.089718
0.01	100	30	no		144.000000	0.068714
0.01	500	30	no		225.000000	0.877403
0.10	5	30	no		138.000000	0.051931
0.10	50	30	yes		101.000000	0.006836
0.10	100	30	no		186.000000	0.338856
0.10	500	30	yes		104.000000	0.008217
0.50	5	30	n/a	*	equal	equal
0.50	50	30	yes		116.000000	0.016566
0.50	100	30	no		183.000000	0.308615
0.50	500	30	yes		97.000000	0.005320
5.00	5	30	no		175.000000	0.236936
5.00	50	30	yes		110.000000	0.011748
5.00	100	30	no		159.000000	0.130592
5.00	500	30	no		142.000000	0.062683
50.00	5	30	yes		134.000000	0.042767
50.00	50	30	no		138.000000	0.051931
50.00	100	30	no		154.000000	0.106394
50.00	500	30	no		146.000000	0.075213

Metric: Accuracy Root Mean Square Error (RMSE) normalized to the lowest one achieved.

C	W	n	SS?	OPTIMAL?	z-stat	p-val
0.01	5	30	no		196.000000	0.452807
0.01	50	30	no		157.000000	0.120445
0.01	100	30	yes		111.000000	0.012453
0.01	500	30	no		224.000000	0.861213
0.10	5	30	no		184.000000	0.318491
0.10	50	30	no		164.000000	0.158855
0.10	100	30	no		200.000000	0.503833

0.10	500	30	no		164.000000	0.158855
0.50	5	30	no		211.000000	0.658331
0.50	50	30	no		224.000000	0.861213
0.50	100	30	no		226.000000	0.893644
0.50	500	30	no		208.000000	0.614315
5.00	5	30	no		225.000000	0.877403
5.00	50	30	yes		136.000000	0.047162
5.00	100	30	n/a	*	equal	equal
5.00	500	30	no		141.000000	0.059836
50.00	5	30	no		151.000000	0.093676
50.00	50	30	no		155.000000	0.110926
50.00	100	30	no		174.000000	0.228880
50.00	500	30	yes		92.000000	0.003854

Metric: Percentage of races that met accuracy target normalized to the highest one achieved.

C	W	n	SS?	OPTIMAL?	z-stat	p-val
0.01	5	30	n/a	*	equal	equal
0.01	50	30	n/a		equal	equal
0.01	100	30	n/a		equal	equal
0.01	500	30	n/a		equal	equal
0.10	5	30	n/a		equal	equal
0.10	50	30	n/a		equal	equal
0.10	100	30	n/a		equal	equal
0.10	500	30	n/a		equal	equal
0.50	5	30	n/a		equal	equal
0.50	50	30	n/a		equal	equal
0.50	100	30	n/a		equal	equal
0.50	500	30	n/a		equal	equal
5.00	5	30	n/a		equal	equal
5.00	50	30	n/a		equal	equal
5.00	100	30	n/a		equal	equal
5.00	500	30	n/a		equal	equal
50.00	5	30	n/a		equal	equal
50.00	50	30	n/a		equal	equal
50.00	100	30	n/a		equal	equal
50.00	500	30	n/a		equal	equal

Metric: Time to convergence (number of races) normalized to the lowest one achieved.

C	W	n	SS?	OPTIMAL?	z-stat	p-val
0.01	5	30	yes		108.500000	0.027917
0.01	50	30	yes		83.500000	0.000000
0.01	100	30	n/a	*	equal	equal
0.01	500	30	yes		100.500000	0.008155

0.10	5	30	yes		176.000000	0.003466
0.10	50	30	yes		184.000000	0.000000
0.10	100	30	yes		93.500000	0.003195
0.10	500	30	no		117.000000	0.058527
0.50	5	30	yes		160.000000	0.010778
0.50	50	30	yes		94.000000	0.003421
0.50	100	30	yes		100.500000	0.002696
0.50	500	30	no		129.500000	0.067406
5.00	5	30	yes		71.500000	0.000805
5.00	50	30	no		143.500000	0.126748
5.00	100	30	yes		90.000000	0.011113
5.00	500	30	yes		98.500000	0.009864
50.00	5	30	yes		131.500000	0.012225
50.00	50	30	yes		98.000000	0.000000
50.00	100	30	yes		128.500000	0.033744
50.00	500	30	yes		115.000000	0.000000

Poisson

Metric: Mean throughput (races/s) normalized to the fastest configuration.

C	W	n	SS?	OPTIMAL?	z-stat	p-val
0.01	5	30	yes		128.000000	0.031603
0.01	50	30	no		183.000000	0.308615
0.01	100	30	no		165.000000	0.165027
0.01	500	30	yes		128.000000	0.031603
0.10	5	30	no		153.000000	0.102011
0.10	50	30	no		213.000000	0.688359
0.10	100	30	no		180.000000	0.280214
0.10	500	30	no		220.000000	0.797098
0.50	5	30	no		141.000000	0.059836
0.50	50	30	no		218.000000	0.765519
0.50	100	30	no		212.000000	0.673280
0.50	500	30	no		226.000000	0.893644
5.00	5	30	yes		131.000000	0.036826
5.00	50	30	n/a	*	equal	equal
5.00	100	30	no		183.000000	0.308615
5.00	500	30	no		222.000000	0.829013
50.00	5	30	no		203.000000	0.544006
50.00	50	30	yes		123.000000	0.024308
50.00	100	30	yes		90.000000	0.003379
50.00	500	30	yes		83.000000	0.002105

Metric: Best candidate throughput (races/s) normalized to the fastest candidate.

C	W	n	SS?	OPTIMAL?	z-stat	p-val
0.01	5	30	yes		6.000000	0.000003
0.01	50	30	yes		14.000000	0.000007
0.01	100	30	yes		22.000000	0.000015
0.01	500	30	yes		57.000000	0.000306
0.10	5	30	yes		15.000000	0.000008
0.10	50	30	yes		0.000000	0.000002
0.10	100	30	yes		25.000000	0.000020
0.10	500	30	yes		40.000000	0.000075
0.50	5	30	yes		2.000000	0.000002
0.50	50	30	yes		1.000000	0.000002
0.50	100	30	yes		9.000000	0.000004
0.50	500	30	yes		36.000000	0.000053
5.00	5	30	yes		2.000000	0.000002
5.00	50	30	yes		2.000000	0.000002
5.00	100	30	yes		17.000000	0.000009
5.00	500	30	yes		50.000000	0.000174
50.00	5	30	yes		27.000000	0.000024
50.00	50	30	yes		39.000000	0.000069
50.00	100	30	yes		77.000000	0.001382
50.00	500	30	n/a	*	equal	equal

Metric: Accuracy Root Mean Square Error (RMSE) normalized to the lowest one achieved.

C	W	n	SS?	OPTIMAL?	z-stat	p-val
0.01	5	30	yes		0.000000	0.000002
0.01	50	30	yes		37.000000	0.000058
0.01	100	30	yes		8.000000	0.000004
0.01	500	30	yes		100.000000	0.006424
0.10	5	30	yes		0.000000	0.000002
0.10	50	30	yes		1.000000	0.000002
0.10	100	30	yes		39.000000	0.000069
0.10	500	30	yes		103.000000	0.007731
0.50	5	30	yes		0.000000	0.000002
0.50	50	30	yes		11.000000	0.000005
0.50	100	30	yes		35.000000	0.000049
0.50	500	30	yes		69.000000	0.000771
5.00	5	30	yes		0.000000	0.000002
5.00	50	30	yes		1.000000	0.000002
5.00	100	30	yes		27.000000	0.000024
5.00	500	30	yes		55.000000	0.000261
50.00	5	30	yes		21.000000	0.000014
50.00	50	30	yes		96.000000	0.004992
50.00	100	30	yes		117.000000	0.017518

50.00	500	30	n/a	*	equal	equal

Metric: Percentage of races that met accuracy target normalized to the highest one achieved.

C	W	n	SS?	OPTIMAL?	z-stat	p-val
0.01	5	30	no		1.000000	0.654721
0.01	50	30	yes		2.000000	0.000276
0.01	100	30	yes		1.000000	0.001225
0.01	500	30	yes		1.000000	0.000352
0.10	5	30	no		1.000000	0.285049
0.10	50	30	yes		1.000000	0.017290
0.10	100	30	yes		0.000000	0.000982
0.10	500	30	yes		1.000000	0.004439
0.50	5	30	n/a	*	equal	equal
0.50	50	30	yes		3.000000	0.012515
0.50	100	30	yes		1.000000	0.027992
0.50	500	30	yes		1.000000	0.017290
5.00	5	30	no		1.000000	0.285049
5.00	50	30	yes		5.000000	0.002865
5.00	100	30	yes		4.000000	0.002329
5.00	500	30	yes		2.000000	0.000276
50.00	5	30	yes		0.000000	0.000038
50.00	50	30	yes		0.000000	0.000012
50.00	100	30	yes		0.000000	0.000002
50.00	500	30	yes		0.000000	0.000004

Metric: Time to convergence (number of races) normalized to the lowest one achieved.

C	W	n	SS?	OPTIMAL?	z-stat	p-val
0.01	5	30	yes		45.000000	0.000000
0.01	50	30	yes		26.000000	0.000000
0.01	100	30	yes		51.000000	0.000000
0.01	500	30	yes		35.000000	0.000000
0.10	5	30	yes		16.500000	0.000000
0.10	50	30	yes		19.500000	0.000000
0.10	100	30	yes		37.500000	0.000000
0.10	500	30	yes		42.000000	0.000000
0.50	5	30	yes		20.000000	0.000000
0.50	50	30	yes		26.000000	0.000000
0.50	100	30	yes		15.000000	0.000000
0.50	500	30	yes		30.000000	0.000000
5.00	5	30	yes		24.000000	0.000000
5.00	50	30	yes		18.000000	0.000000
5.00	100	30	n/a	*	equal	equal
5.00	500	30	yes		9.000000	0.000000

50.00	5	30	yes		37.500000	0.000000
50.00	50	30	yes		35.000000	0.000000
50.00	100	30	yes		24.000000	0.000000
50.00	500	30	yes		0.000000	0.000047

Binpacking

Metric: Mean throughput (races/s) normalized to the fastest configuration.

C	W	n	SS?	OPTIMAL?	z-stat	p-val
0.01	5	30	yes		12.000000	0.000006
0.01	50	30	no		226.000000	0.893644
0.01	100	30	no		230.000000	0.958990
0.01	500	30	no		229.000000	0.942611
0.10	5	30	yes		12.000000	0.000006
0.10	50	30	no		214.000000	0.703564
0.10	100	30	no		212.000000	0.673280
0.10	500	30	no		199.000000	0.490798
0.50	5	30	yes		12.000000	0.000006
0.50	50	30	n/a	*	equal	equal
0.50	100	30	no		209.000000	0.628843
0.50	500	30	no		203.000000	0.544006
5.00	5	30	yes		3.000000	0.000002
5.00	50	30	no		182.000000	0.298944
5.00	100	30	no		193.000000	0.416534
5.00	500	30	yes		130.000000	0.035009
50.00	5	30	yes		34.000000	0.000044
50.00	50	30	yes		67.000000	0.000664
50.00	100	30	yes		95.000000	0.004682
50.00	500	30	yes		45.000000	0.000115

Metric: Best candidate throughput (races/s) normalized to the fastest candidate.

C	W	n	SS?	OPTIMAL?	z-stat	p-val
0.01	5	8	n/a	*	equal	equal
0.01	50	30	yes		0.000000	0.011719
0.01	100	29	yes		1.000000	0.017290
0.01	500	30	yes		0.000000	0.011719
0.10	5	14	no		11.000000	0.326989
0.10	50	30	yes		0.000000	0.011719
0.10	100	30	yes		0.000000	0.011719
0.10	500	30	yes		0.000000	0.011719
0.50	5	10	no		10.000000	0.262618
0.50	50	29	yes		0.000000	0.011719

0.50	100	28	yes		0.000000	0.011719
0.50	500	28	yes		0.000000	0.011719
5.00	5	10	no		6.000000	0.092892
5.00	50	29	yes		4.000000	0.049950
5.00	100	30	yes		1.000000	0.017290
5.00	500	28	yes		0.000000	0.011719
50.00	5	28	no		15.000000	0.674424
50.00	50	30	yes		0.000000	0.011719
50.00	100	30	yes		0.000000	0.011719
50.00	500	30	yes		0.000000	0.011719

Metric: Accuracy Root Mean Square Error (RMSE) normalized to the lowest one achieved.

C	W	n	SS?	OPTIMAL?	z-stat	p-val
0.01	5	30	n/a	*	equal	equal
0.01	50	30	yes		0.000000	0.000002
0.01	100	30	yes		4.000000	0.000003
0.01	500	30	yes		0.000000	0.000002
0.10	5	30	no		190.000000	0.382034
0.10	50	30	yes		0.000000	0.000002
0.10	100	30	yes		0.000000	0.000002
0.10	500	30	yes		0.000000	0.000002
0.50	5	30	no		230.000000	0.958990
0.50	50	30	yes		7.000000	0.000004
0.50	100	30	yes		2.000000	0.000002
0.50	500	30	yes		3.000000	0.000002
5.00	5	30	no		232.000000	0.991795
5.00	50	30	yes		3.000000	0.000002
5.00	100	30	yes		0.000000	0.000002
5.00	500	30	yes		1.000000	0.000002
50.00	5	30	yes		0.000000	0.000002
50.00	50	30	yes		0.000000	0.000002
50.00	100	30	yes		0.000000	0.000002
50.00	500	30	yes		0.000000	0.000002

Metric: Percentage of races that met accuracy target normalized to the highest one achieved.

C	W	n	SS?	OPTIMAL?	z-stat	p-val
0.01	5	30	yes		33.000000	0.000108
0.01	50	30	yes		104.000000	0.008217
0.01	100	30	yes		82.000000	0.001965
0.01	500	30	yes		80.000000	0.001709
0.10	5	30	yes		74.000000	0.003309
0.10	50	30	yes		88.000000	0.002957
0.10	100	30	yes		105.000000	0.008730

0.10	500	30	yes		82.000000	0.001965
0.50	5	30	yes		45.000000	0.000191
0.50	50	30	yes		101.000000	0.006836
0.50	100	30	yes		57.000000	0.000306
0.50	500	30	yes		74.000000	0.001114
5.00	5	30	yes		36.000000	0.000143
5.00	50	30	yes		80.000000	0.001709
5.00	100	30	yes		67.000000	0.000664
5.00	500	30	yes		41.000000	0.000082
50.00	5	30	n/a	*	equal	equal
50.00	50	30	no		163.000000	0.152861
50.00	100	30	yes		112.000000	0.013194
50.00	500	30	yes		135.000000	0.044919

Metric: Time to convergence (number of races) normalized to the lowest one achieved.

C	W	n	SS?	OPTIMAL?	z-stat	p-val
0.01	5	8	yes		1.000000	0.017290
0.01	50	30	no		162.000000	0.335713
0.01	100	29	no		185.500000	0.513149
0.01	500	30	no		178.500000	0.389474
0.10	5	14	yes		6.000000	0.004451
0.10	50	30	yes		129.500000	0.047519
0.10	100	30	no		184.500000	0.491624
0.10	500	30	no		185.000000	0.923441
0.50	5	10	yes		4.000000	0.016605
0.50	50	29	no		197.000000	0.799749
0.50	100	28	n/a	*	equal	equal
0.50	500	28	no		165.000000	0.379495
5.00	5	10	yes		4.000000	0.028402
5.00	50	29	yes		101.000000	0.020196
5.00	100	30	yes		126.500000	0.040749
5.00	500	28	no		189.500000	0.596802
50.00	5	28	yes		12.000000	0.000014
50.00	50	30	no		145.000000	0.095697
50.00	100	30	no		181.500000	0.742895
50.00	500	30	no		165.000000	0.224598

A.2.2 AMD48

Sort

Metric: Mean throughput (races/s) normalized to the fastest configuration.

C	W	n	SS?	OPTIMAL?	z-stat	p-val
0.01	5	20	yes		0.000000	0.000089
0.01	50	20	yes		18.000000	0.001162
0.01	100	20	yes		34.000000	0.008034
0.01	500	20	yes		3.000000	0.000140
0.10	5	20	yes		0.000000	0.000089
0.10	50	20	yes		1.000000	0.000103
0.10	100	20	yes		3.000000	0.000140
0.10	500	20	yes		11.000000	0.000449
0.50	5	20	yes		1.000000	0.000103
0.50	50	20	yes		9.000000	0.000338
0.50	100	20	yes		5.000000	0.000189
0.50	500	20	yes		13.000000	0.000593
5.00	5	20	yes		0.000000	0.000089
5.00	50	20	yes		28.000000	0.004045
5.00	100	20	no		93.000000	0.654159
5.00	500	20	no		71.000000	0.204330
50.00	5	20	yes		4.000000	0.000163
50.00	50	20	n/a	*	equal	equal
50.00	100	20	no		62.000000	0.108427
50.00	500	20	yes		24.000000	0.002495

Metric: Best candidate throughput (races/s) normalized to the fastest candidate.

C	W	n	SS?	OPTIMAL?	z-stat	p-val
0.01	5	20	yes		1.000000	0.000103
0.01	50	20	yes		13.000000	0.000593
0.01	100	20	yes		20.000000	0.001507
0.01	500	20	yes		6.000000	0.000219
0.10	5	20	yes		0.000000	0.000089
0.10	50	20	yes		1.000000	0.000103
0.10	100	20	yes		0.000000	0.000089
0.10	500	20	yes		12.000000	0.000517
0.50	5	20	yes		1.000000	0.000103
0.50	50	20	yes		3.000000	0.000140
0.50	100	20	yes		5.000000	0.000189
0.50	500	20	yes		12.000000	0.000517
5.00	5	20	yes		0.000000	0.000089
5.00	50	20	yes		19.000000	0.001325
5.00	100	20	no		61.000000	0.100458
5.00	500	20	yes		32.000000	0.006425
50.00	5	20	yes		4.000000	0.000163
50.00	50	20	n/a	*	equal	equal
50.00	100	20	no		88.000000	0.525653
50.00	500	20	yes		32.000000	0.006425

Metric: Time to convergence (number of races) normalized to the lowest one achieved.

C	W	n	SS?	OPTIMAL?	z-stat	p-val
0.01	5	20	no		61.000000	0.061360
0.01	50	20	yes		58.000000	0.049374
0.01	100	20	yes		35.500000	0.007955
0.01	500	20	no		78.000000	0.327265
0.10	5	20	no		100.500000	0.753137
0.10	50	20	no		54.500000	0.110449
0.10	100	20	no		66.000000	0.087121
0.10	500	20	yes		21.000000	0.002902
0.50	5	20	no		72.000000	0.122732
0.50	50	20	yes		34.000000	0.006545
0.50	100	20	no		68.000000	0.099859
0.50	500	20	yes		42.500000	0.024955
5.00	5	20	n/a	*	equal	equal
5.00	50	20	yes		21.000000	0.001713
5.00	100	20	yes		8.000000	0.000293
5.00	500	20	yes		8.000000	0.000382
50.00	5	20	yes		1.500000	0.000155
50.00	50	20	yes		0.000000	0.000129
50.00	100	20	yes		0.000000	0.000089
50.00	500	20	yes		0.000000	0.000125

Matrix Approximation

Metric: Mean throughput (races/s) normalized to the fastest configuration.

C	W	n	SS?	OPTIMAL?	z-stat	p-val
0.01	5	20	no		96.000000	0.736875
0.01	50	20	no		75.000000	0.262722
0.01	100	20	no		91.000000	0.601213
0.01	500	20	no		67.000000	0.156004
0.10	5	20	no		98.000000	0.793839
0.10	50	20	no		65.000000	0.135357
0.10	100	20	no		99.000000	0.822760
0.10	500	20	no		68.000000	0.167184
0.50	5	20	no		102.000000	0.910825
0.50	50	20	no		77.000000	0.295878
0.50	100	20	no		103.000000	0.940481
0.50	500	20	no		80.000000	0.350656
5.00	5	20	no		103.000000	0.940481
5.00	50	20	yes		50.000000	0.040044

5.00	100	20	no		77.000000	0.295878
5.00	500	20	no		79.000000	0.331723
50.00	5	20	no		67.000000	0.156004
50.00	50	20	n/a	*	equal	equal
50.00	100	20	yes		47.000000	0.030365
50.00	500	20	no		62.000000	0.108427

Metric: Best candidate throughput (races/s) normalized to the fastest candidate.

C	W	n	SS?	OPTIMAL?	z-stat	p-val
0.01	5	20	no		88.000000	0.525653
0.01	50	20	yes		47.000000	0.030365
0.01	100	20	yes		51.000000	0.043804
0.01	500	20	yes		46.000000	0.027621
0.10	5	20	no		73.000000	0.232226
0.10	50	20	yes		51.000000	0.043804
0.10	100	20	yes		35.000000	0.008968
0.10	500	20	yes		20.000000	0.001507
0.50	5	20	no		73.000000	0.232226
0.50	50	20	no		53.000000	0.052222
0.50	100	20	yes		36.000000	0.009996
0.50	500	20	yes		31.000000	0.005734
5.00	5	20	n/a	*	equal	equal
5.00	50	20	yes		47.000000	0.030365
5.00	100	20	yes		45.000000	0.025094
5.00	500	20	no		67.000000	0.156004
50.00	5	20	yes		34.000000	0.008034
50.00	50	20	no		78.000000	0.313463
50.00	100	20	no		59.000000	0.085924
50.00	500	20	no		57.000000	0.073138

Metric: Accuracy Root Mean Square Error (RMSE) normalized to the lowest one achieved.

C	W	n	SS?	OPTIMAL?	z-stat	p-val
0.01	5	20	no		103.000000	0.940481
0.01	50	20	no		60.000000	0.092963
0.01	100	20	no		81.000000	0.370261
0.01	500	20	yes		52.000000	0.047858
0.10	5	20	no		81.000000	0.370261
0.10	50	20	no		59.000000	0.085924
0.10	100	20	no		100.000000	0.851925
0.10	500	20	no		92.000000	0.627446
0.50	5	20	no		98.000000	0.793839
0.50	50	20	no		104.000000	0.970220
0.50	100	20	no		82.000000	0.390533

0.50	500	20	no		91.000000	0.601213
5.00	5	20	no		79.000000	0.331723
5.00	50	20	yes		43.000000	0.020633
5.00	100	20	no		64.000000	0.125859
5.00	500	20	no		100.000000	0.851925
50.00	5	20	yes		51.000000	0.043804
50.00	50	20	no		53.000000	0.052222
50.00	100	20	n/a	*	equal	equal
50.00	500	20	no		58.000000	0.079322

Metric: Percentage of races that met accuracy target normalized to the highest one achieved.

C	W	n	SS?	OPTIMAL?	z-stat	p-val
0.01	5	20	n/a	*	equal	equal
0.01	50	20	n/a		equal	equal
0.01	100	20	n/a		equal	equal
0.01	500	20	n/a		equal	equal
0.10	5	20	n/a		equal	equal
0.10	50	20	n/a		equal	equal
0.10	100	20	n/a		equal	equal
0.10	500	20	n/a		equal	equal
0.50	5	20	n/a		equal	equal
0.50	50	20	n/a		equal	equal
0.50	100	20	n/a		equal	equal
0.50	500	20	n/a		equal	equal
5.00	5	20	n/a		equal	equal
5.00	50	20	n/a		equal	equal
5.00	100	20	n/a		equal	equal
5.00	500	20	n/a		equal	equal
50.00	5	20	n/a		equal	equal
50.00	50	20	n/a		equal	equal
50.00	100	20	n/a		equal	equal
50.00	500	20	n/a		equal	equal

Metric: Time to convergence (number of races) normalized to the lowest one achieved.

C	W	n	SS?	OPTIMAL?	z-stat	p-val
0.01	5	20	yes		25.000000	0.000000
0.01	50	20	yes		3.000000	0.000000
0.01	100	20	yes		7.000000	0.000000
0.01	500	20	yes		9.000000	0.000000
0.10	5	20	yes		2.000000	0.011719
0.10	50	20	yes		15.500000	0.000000
0.10	100	20	yes		8.000000	0.000000
0.10	500	20	yes		14.000000	0.000000

0.50	5	20	yes		10.500000	0.000000
0.50	50	20	yes		7.000000	0.000000
0.50	100	20	yes		5.000000	0.000000
0.50	500	20	yes		16.000000	0.000000
5.00	5	20	yes		13.500000	0.000000
5.00	50	20	n/a	*	equal	equal
5.00	100	20	yes		8.000000	0.000000
5.00	500	20	yes		18.000000	0.000000
50.00	5	20	yes		6.000000	0.000000
50.00	50	20	yes		7.500000	0.000000
50.00	100	20	yes		13.500000	0.000000
50.00	500	20	yes		13.500000	0.000000

Poisson

Metric: Mean throughput (races/s) normalized to the fastest configuration.

C	W	n	SS?	OPTIMAL?	z-stat	p-val
0.01	5	20	yes		1.000000	0.000103
0.01	50	20	no		84.000000	0.433048
0.01	100	20	no		82.000000	0.390533
0.01	500	20	no		101.000000	0.881293
0.10	5	20	yes		1.000000	0.000103
0.10	50	20	no		70.000000	0.191334
0.10	100	20	no		66.000000	0.145400
0.10	500	20	no		94.000000	0.681322
0.50	5	20	yes		5.000000	0.000189
0.50	50	20	yes		33.000000	0.007189
0.50	100	20	no		65.000000	0.135357
0.50	500	20	yes		36.000000	0.009996
5.00	5	20	yes		14.000000	0.000681
5.00	50	20	no		60.000000	0.092963
5.00	100	20	yes		52.000000	0.047858
5.00	500	20	n/a	*	equal	equal
50.00	5	20	yes		43.000000	0.020633
50.00	50	20	no		74.000000	0.247145
50.00	100	20	no		89.000000	0.550292
50.00	500	20	no		85.000000	0.455273

Metric: Best candidate throughput (races/s) normalized to the fastest candidate.

C	W	n	SS?	OPTIMAL?	z-stat	p-val
0.01	5	20	yes		49.000000	0.036561
0.01	50	20	yes		46.000000	0.027621

0.01	100	20	yes		36.000000	0.009996
0.01	500	20	no		63.000000	0.116888
0.10	5	20	yes		21.000000	0.001713
0.10	50	20	yes		30.000000	0.005111
0.10	100	20	no		56.000000	0.067355
0.10	500	20	yes		51.000000	0.043804
0.50	5	20	yes		4.000000	0.000163
0.50	50	20	yes		34.000000	0.008034
0.50	100	20	yes		27.000000	0.003592
0.50	500	20	yes		35.000000	0.008968
5.00	5	20	yes		19.000000	0.001325
5.00	50	20	yes		49.000000	0.036561
5.00	100	20	yes		26.000000	0.003185
5.00	500	20	yes		40.000000	0.015240
50.00	5	20	yes		34.000000	0.008034
50.00	50	20	no		69.000000	0.178956
50.00	100	20	yes		40.000000	0.015240
50.00	500	20	n/a	*	equal	equal

Metric: Accuracy Root Mean Square Error (RMSE) normalized to the lowest one achieved.

C	W	n	SS?	OPTIMAL?	z-stat	p-val
0.01	5	20	yes		0.000000	0.000089
0.01	50	20	yes		43.000000	0.020633
0.01	100	20	yes		36.000000	0.009996
0.01	500	20	no		68.000000	0.167184
0.10	5	20	yes		0.000000	0.000089
0.10	50	20	yes		25.000000	0.002821
0.10	100	20	yes		13.000000	0.000593
0.10	500	20	yes		50.000000	0.040044
0.50	5	20	yes		0.000000	0.000089
0.50	50	20	yes		15.000000	0.000780
0.50	100	20	yes		35.000000	0.008968
0.50	500	20	yes		13.000000	0.000593
5.00	5	20	yes		0.000000	0.000089
5.00	50	20	yes		27.000000	0.003592
5.00	100	20	yes		14.000000	0.000681
5.00	500	20	yes		40.000000	0.015240
50.00	5	20	yes		0.000000	0.000089
50.00	50	20	no		83.000000	0.411465
50.00	100	20	no		76.000000	0.278965
50.00	500	20	n/a	*	equal	equal

Metric: Percentage of races that met accuracy target normalized to the highest one achieved.

C	W	n	SS?	OPTIMAL?	z-stat	p-val
0.01	5	20	no		5.000000	0.500184
0.01	50	20	yes		0.000000	0.005062
0.01	100	20	yes		2.000000	0.005847
0.01	500	20	yes		2.000000	0.005847
0.10	5	20	n/a	*	equal	equal
0.10	50	20	yes		0.000000	0.007686
0.10	100	20	yes		2.000000	0.025062
0.10	500	20	yes		0.000000	0.005062
0.50	5	20	no		5.000000	0.351076
0.50	50	20	no		0.000000	0.067889
0.50	100	20	yes		2.000000	0.025062
0.50	500	20	no		1.000000	0.079616
5.00	5	20	no		3.000000	0.465209
5.00	50	20	yes		6.000000	0.028417
5.00	100	20	yes		5.000000	0.004649
5.00	500	20	yes		0.000000	0.000196
50.00	5	20	yes		1.000000	0.001140
50.00	50	20	yes		0.000000	0.000129
50.00	100	20	yes		0.000000	0.000089
50.00	500	20	yes		0.000000	0.000132

Metric: Time to convergence (number of races) normalized to the lowest one achieved.

C	W	n	SS?	OPTIMAL?	z-stat	p-val
0.01	5	20	yes		28.000000	0.005143
0.01	50	20	yes		46.000000	0.013828
0.01	100	20	yes		63.000000	0.040437
0.01	500	20	yes		26.500000	0.009243
0.10	5	20	no		68.500000	0.294692
0.10	50	20	no		72.500000	0.151243
0.10	100	20	no		75.000000	0.412056
0.10	500	20	yes		50.000000	0.039583
0.50	5	20	no		53.000000	0.508737
0.50	50	20	no		43.000000	0.071637
0.50	100	20	no		83.000000	0.828995
0.50	500	20	no		50.500000	0.080834
5.00	5	20	n/a	*	equal	equal
5.00	50	20	yes		56.500000	0.039641
5.00	100	20	no		76.000000	0.490595
5.00	500	20	yes		44.000000	0.037965
50.00	5	20	no		70.000000	0.330914
50.00	50	20	yes		0.000000	0.000185
50.00	100	20	yes		37.000000	0.043336
50.00	500	20	yes		25.500000	0.004982

Binpacking

Metric: Mean throughput (races/s) normalized to the fastest configuration.

C	W	n	SS?	OPTIMAL?	z-stat	p-val
0.01	5	20	yes		1.000000	0.000103
0.01	50	20	yes		51.000000	0.043804
0.01	100	20	no		63.000000	0.116888
0.01	500	20	no		74.000000	0.247145
0.10	5	20	yes		1.000000	0.000103
0.10	50	20	no		84.000000	0.433048
0.10	100	20	no		90.000000	0.575486
0.10	500	20	no		82.000000	0.390533
0.50	5	20	yes		0.000000	0.000089
0.50	50	20	no		79.000000	0.331723
0.50	100	20	no		71.000000	0.204330
0.50	500	20	no		74.000000	0.247145
5.00	5	20	yes		1.000000	0.000103
5.00	50	20	no		87.000000	0.501591
5.00	100	20	no		62.000000	0.108427
5.00	500	20	n/a	*	equal	equal
50.00	5	20	yes		7.000000	0.000254
50.00	50	20	yes		11.000000	0.000449
50.00	100	20	yes		12.000000	0.000517
50.00	500	20	yes		11.000000	0.000449

Metric: Best candidate throughput (races/s) normalized to the fastest candidate.

C	W	n	SS?	OPTIMAL?	z-stat	p-val
0.01	5	7	no		3.000000	0.224916
0.01	50	20	yes		0.000000	0.043114
0.01	100	19	yes		0.000000	0.043114
0.01	500	19	yes		0.000000	0.043114
0.10	5	5	n/a	*	equal	equal
0.10	50	20	yes		0.000000	0.043114
0.10	100	19	yes		0.000000	0.043114
0.10	500	19	yes		0.000000	0.043114
0.50	5	5	no		6.000000	0.685830
0.50	50	19	no		2.000000	0.138011
0.50	100	19	yes		0.000000	0.043114
0.50	500	19	yes		0.000000	0.043114
5.00	5	12	no		6.000000	0.685830
5.00	50	19	no		1.000000	0.079616

5.00	100	20	yes		0.000000	0.043114
5.00	500	19	yes		0.000000	0.043114
50.00	5	20	no		3.000000	0.224916
50.00	50	20	yes		0.000000	0.043114
50.00	100	20	yes		0.000000	0.043114
50.00	500	20	yes		0.000000	0.043114

Metric: Accuracy Root Mean Square Error (RMSE) normalized to the lowest one achieved.

C	W	n	SS?	OPTIMAL?	z-stat	p-val
0.01	5	20	no		94.000000	0.681322
0.01	50	20	yes		0.000000	0.000089
0.01	100	20	yes		0.000000	0.000089
0.01	500	20	yes		0.000000	0.000089
0.10	5	20	no		94.000000	0.681322
0.10	50	20	yes		0.000000	0.000089
0.10	100	20	yes		2.000000	0.000120
0.10	500	20	yes		3.000000	0.000140
0.50	5	20	n/a	*	equal	equal
0.50	50	20	yes		0.000000	0.000089
0.50	100	20	yes		0.000000	0.000089
0.50	500	20	yes		3.000000	0.000140
5.00	5	20	no		59.000000	0.085924
5.00	50	20	yes		0.000000	0.000089
5.00	100	20	yes		0.000000	0.000089
5.00	500	20	yes		0.000000	0.000089
50.00	5	20	yes		0.000000	0.000089
50.00	50	20	yes		0.000000	0.000089
50.00	100	20	yes		0.000000	0.000089
50.00	500	20	yes		0.000000	0.000089

Metric: Percentage of races that met accuracy target normalized to the highest one achieved.

C	W	n	SS?	OPTIMAL?	z-stat	p-val
0.01	5	20	yes		24.000000	0.002495
0.01	50	20	yes		35.000000	0.008968
0.01	100	20	yes		5.000000	0.000189
0.01	500	20	yes		15.000000	0.000780
0.10	5	20	yes		12.000000	0.000517
0.10	50	20	yes		23.000000	0.002204
0.10	100	20	yes		36.000000	0.009996
0.10	500	20	yes		3.000000	0.000140
0.50	5	20	yes		10.000000	0.000390
0.50	50	20	yes		44.000000	0.022769
0.50	100	20	yes		39.000000	0.013741

0.50	500	20	yes		11.000000	0.000449
5.00	5	20	no		69.000000	0.178956
5.00	50	20	yes		27.000000	0.003592
5.00	100	20	yes		8.000000	0.000293
5.00	500	20	yes		15.000000	0.000780
50.00	5	20	no		77.000000	0.295878
50.00	50	20	no		81.000000	0.370261
50.00	100	20	no		103.000000	0.940481
50.00	500	20	n/a	*	equal	equal

Metric: Time to convergence (number of races) normalized to the lowest one achieved.

C	W	n	SS?	OPTIMAL?	z-stat	p-val
0.01	5	7	no		3.000000	0.062979
0.01	50	20	yes		23.000000	0.003762
0.01	100	19	no		55.000000	0.067149
0.01	500	19	no		93.000000	0.877421
0.10	5	5	yes		0.000000	0.043114
0.10	50	20	no		64.000000	0.126504
0.10	100	19	no		80.000000	0.546088
0.10	500	19	yes		40.000000	0.047531
0.50	5	5	yes		0.000000	0.043114
0.50	50	19	no		70.500000	0.202373
0.50	100	19	no		63.500000	0.124936
0.50	500	19	n/a	*	equal	equal
5.00	5	12	yes		0.000000	0.002218
5.00	50	19	no		41.000000	0.052624
5.00	100	20	no		71.000000	0.211264
5.00	500	19	no		64.500000	0.233256
50.00	5	20	yes		33.500000	0.011088
50.00	50	20	no		79.500000	0.368853
50.00	100	20	no		74.500000	0.261956
50.00	500	20	no		67.000000	0.270185