DeepTuner: A System for Search Technique Recommendation in Program Autotuning

by

Kevin Wu

B.S., Massachusetts Institute of Technology (2014)

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Masters of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

July 2015

The author hereby grants to M.I.T. permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole and in part in any medium now known or hereafter created.

Author: ....

Department of Electrical Engineering and Computer Science

July 17, 2015

Certified by: ....

Kalyan Veeramachaneni, Thesis Supervisor

July 17, 2015

Certified by: ....

Prof. Saman Amarasinghe, Thesis Supervisor

July 17, 2015

Accepted by: ....

Prof. Albert R. Meyer, Chairman, Masters of Engineering Thesis Committee

DeepTuner: A System for Search Technique Recommendation in Program Autotuning

by

Kevin Wu

Submitted to the Department of Electrical Engineering and Computer Science

July 17, 2015

in partial fulfilment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

Abstract

OpenTuner can help users achieve better or more portable performance in their specific domain through program autotuning. A key challenge for users seeking good autotuning performance in OpenTuner is selecting a search approach appropriate for problem. However, not only are current in-situ learning search approaches not robust enough to handle all search spaces, but there are also too many possible search approaches for a user to examine manually after factoring in composable techniques.

In this thesis, we introduce DeepTuner, a system for search approach recommendation operating across OpenTuner autotuning sessions to facilitate development of robust transfer learning search approaches. By utilizing historical autotuning data via DeepTuner's technique recommendation endpoints, the new search approaches can efficiently explore the space of possible search approaches and the autotuning space simultaneously, resulting in an adaptive, self-improving search approach.

We demonstrate the robustness that recommendation brings on nine problems spread over three domains for a variety of initial technique sets. In particular, we show that the new *Database Initialized Recommendation Bandit Meta-technique* is highly robust, performing on par or significantly better than various old in-situ search approaches in OpenTuner. We achieve up to 3.7x performance improvement over the old default in-situ search approach for OpenTuner in the TSP domain.

Acknowledgements

First of all, I'd like to thank my advisors, Saman Amarasinghe and Kalyan Veeramachaneni. Throughout his many roles as an undergraduate advisor, PI, and thesis advisor to me, Saman has consistently provided great ideas and sound advice to keep me on track. His guidance and vision have been indispensable in breaking through (or simply walking around) the roadblocks hindering my progress. I also couldn't have asked for more out of an advisor with Kalyan. His knowledge and experience, combined with a tireless dedication to his projects and advisees, made working with him an absolute pleasure. His easy going personality and willingness to share new tools with me to make my life easier were the icing on the cake. This thesis definitely wouldn't have been possible without the support of these two amazing individuals and the great feedback they've given me. I'm truly grateful for all for their time and efforts.

Next, I'd like to thank the many talented members of the COMMIT group - past and present - for making my CSAIL experience great. Particular affirmations to Jason for his patience with helping me understand the ins and outs of OpenTuner; Jeffrey for many keen insights, his no-nonsense practical advice, and a uniquely dry sense of humour; Shoaib for help working with so many different languages and systems, despite being one of the busiest people I've ever met; Charith for some fun times with Helium; and finally my fellow MEng-ers Giuseppe and Yoana.

Also, I want to thank MIT for being the awesome, intellectually stimulating, open-to-anything and supportive community that enabled me to pursue so many of my different interests. I don't think anywhere else could have put up with the academic ADHD of my undergraduate years. I couldn't have asked for a better place to learn. Along those lines, here's a shout-out to all the great friends I've made at MIT. MIT would have been 99.9% less enjoyable without all of you brilliant people!

Last but not least, I'd like to thank my family. Without their unwavering support, I wouldn't have been able to focus on and enjoy my experience at MIT to the extent that I did. I feel truly blessed to have been raised by such devoted parents.

Contents

List of Figures **6**

List of Tables **8**

1 Introduction **9**

1.1 Program Autotuning . . . 9

1.2 OpenTuner . . . 10

1.3 DeepTuner . . . 12

1.4 Contributions . . . 16

1.5 Organization . . . 17

2 Related Work **18**

2.1 Autotuning Frameworks . . . 18

2.1.1 Linear Algebra . . . 18

2.1.2 Signal Processing . . . 19

2.1.3 Stencil Computations . . . 19

2.1.4 Runtime Systems . . . 20

2.2 Search Technique Hybridization . . . 20

2.3 Algorithm Recommendation . . . 21

3 The OpenTuner Framework **22**

3.1 Problem Definition . . . 22

3.2 Search Approach . . . 24

3.2.1 AUC Bandit Meta-Technique . . . 24

4 Search Technique Composition **26**

4.1 Creating a Technique Base . . . 27

4.2 Choosing a Set of Operators . . . 28
4.2.1 Parameter Types . . . 30
4.2.2 Available Operators . . . 30

5 The Cross-session Database **36**
5.1 Database Schema . . . 36
5.1.1 Storing Representation Parameter Structure . . . 37
5.2 Encoding Search Approach Data . . . 37
5.2.1 Tracking Hyper-parameter Information . . . 38
5.2.2 Operator Maps . . . 38
5.3 Uploading Tuning Data . . . 39
5.4 Historical Tuning Run Data Generation . . . 39
5.4.1 Using the Cross-session Database . . . 39

6 A Recommender System **43**
6.1 The Recommendation Bandit Meta-Technique . . . 45
6.2 An Example of the Recommendation Process . . . 49
6.3 Advantages of the Recommendation Bandit Meta-Technique . . . 51

7 Results **52**
7.1 Testing Protocol . . . 52
7.1.1 Meta-techniques Tested . . . 52
7.2 Search Approach Performance Comparisons . . . 54
7.2.1 Halide . . . 55
7.2.2 Petabricks . . . 55
7.2.3 Travelling Salesman Problem . . . 62
7.3 Key Findings . . . 80

8 Future Work **82**
8.1 Expanding Recommendation in DeepTuner . . . 82
8.2 Search Affinity Metrics . . . 82
8.3 Other Modes for Transfer Learning . . . 83
8.4 Addressing Privacy Concerns . . . 83
8.5 Developing New Parameter Types and Operators . . . 83

9 Conclusion **84**

Bibliography **85**

A Appendix A: Definitions and Terminology **90**

A.1 Basic Autotuning Terminology 90

A.2 Autotuner Specification 91

A.3 Search Approach 92

A.4 Search and Evaluation 94

A.5 Recommender System 94

List of Figures

1.1 Levels of learning in OpenTuner search approaches . . . 13
1.2 Autotuning performance of an in-situ learning vs. a transfer learning search approach . . . 16
3.1 Structure of a problem definition . . . 23
4.1 Hierarchy of OpenTuner's built-in parameter types . . . 31
5.1 Database schema for the cross-session database . . . 42
6.1 Specifying a search approach in OpenTuner . . . 44
6.2 Search process for the the recommendation bandit meta-technique . . . 48
7.1 Search approach performance in Halide bilateral grid by initial technique set . . . 56
7.2 End of autotuning execution times for the Halide bilateral grid benchmark . . . 57
7.3 Search approach performance in the Halide bilateral grid benchmark . . . 58
7.4 Search approach performance in Petabricks matrix approximation by initial technique set . . . 59
7.5 End of autotuning execution times for the Petabricks matrix approximation benchmark . . . 60
7.6 Search approach performance in the Petabricks matrix approximation benchmark . . . 61
7.7 Search approach performance in Petabricks Strassen by initial technique set . . . 63
7.8 Search approach performance in the Petabricks Strassen benchmark . . . 64
7.9 Search approach performance in TSP att48 by initial technique set . . . 65
7.10 Search approach performance in TSP dantzig42 by initial technique set . . . 66
7.11 Search approach performance in TSP att48 . . . 68
7.12 End of autotuning best route lengths for the TSP att48 benchmark . . . 69
7.13 Search approach performance in TSP dantzig42 . . . 70
7.14 End of autotuning best route lengths for the TSP dantzig42 benchmark . . . 71
7.15 Search approach performance in ATSP ftv44 and ftv170 . . . 73

7.16 End of autotuning best route lengths for the ATSP ftv44 and ftv170 benchmarks 74
7.17 Database initialized search approaches in the ATSP ftv44 and ftv170 benchmarks 75
7.18 Search approach performance in the ATSP rbg323 representation 76
7.19 End of autotuning best route lengths for the ATSP rbg323 benchmark 77
7.20 Search approach performance in the ATSP p43 representation 78
7.21 End of autotuning best route lengths for the ATSP p43 benchmark 79

List of Tables

1.1 Number of operators for parameter types in the Halide domain 11

1.2 Aggregate search affinity for technique classes on the brazil58 TSP benchmark 14

4.1 Implemented technique bases in OpenTuner 29

4.2 Descriptions of parameter types in OpenTuner 30

4.3 Available operators in OpenTuner by parameter type 33

4.4 Available operators in OpenTuner by parameter type (continued) 34

4.5 Available operators in OpenTuner by parameter type (continued) 35

5.1 Tuning run data contained in the cross-session database by representation 40

6.1 Example performance data in the cross-session database 49

6.2 Example search affinity similarity between representations in the cross-session database and att48 50

7.1 Labels for tested search approaches 54

7.2 Benchmarks in the TSP autotuner 67

7.3 Relative performance of bndt-A and rec-bndt-dbi on TSP benchmarks 81

Chapter 1

Introduction

1.1 Program Autotuning

Program autotuning has seen increased use in performance-critical domains as a supplement - or even replacement - for traditional human-guided optimization of programs. By delegating the exploration of the space of possible optimizations or alternate implementations to an automated search process, a programmer can save the time and effort of having to manually generate and test program variants. This also allows for the search of larger spaces. In addition, autotuning increases performance portability of programs. An autotuner can be easily re-run to determine new optimal settings when the underlying architecture or environment changes, unlike with hand-tuning where an experienced programmer must repeat the entire hand-tuning process again.

Despite increased use of autotuners in specific projects, the development of general, project-agnostic autotuning frameworks has been much more difficult. This is due to the combination of several properties of problem search spaces.

First, is the size of the search space. Autotuning search spaces are often exponentially large due to the combinatorial nature of optimization choices, making exhaustive search intractable. It is possible to prune the search space and artificially constrain its size. However, this introduces programmer bias in terms of which configurations will be examined, so excessive pruning can easily result in overlooking good configurations that were not intuitive to the programmer. Thus, having an intelligent machine learning search approach is critical to discover good configurations in these massive search spaces within a small number of evaluations.

However, autotuning search spaces are highly diverse in topography and complexity, both of which affect the way a particular technique interacts with the space. For example, for some simple search spaces that are

relatively smooth and monotonic, a hill climbing technique would perform reasonably well. But in practice, search spaces can have multiple hills, discontinuities, plateaus, and other complexities like interdependencies or coupling between parameters. As a result, there is no guarantee that the same hill-climbing technique, or any technique for that matter, will perform well on a different, arbitrarily chosen problem. To complicate matters further though, the same problem can often be projected into multiple different *parameters in the representation* (see Appendix A.2), resulting in wildly different search spaces. If the selected search approach interacts too poorly with the parameters in the representation, then the autotuner will be unable to find good configurations in a reasonable amount of time.

Thus, the key challenge for making a general autotuning framework lies in being able to select a search approach that can efficiently search a representation that is arbitrarily defined by the user. And unfortunately, taking the naive search approach of just using a single technique is highly insufficient.

1.2 OpenTuner

OpenTuner provides a general framework for domain-specific autotuner creation that tackles this challenge [5]. OpenTuner features a library of extensible parameters that allows for the support of the arbitrary data-types are required to define a representation for a problem. An extensible library of techniques in OpenTuner helps to ensure that for any representation, there will be a technique that performs reasonably well.

Meanwhile, OpenTuner improves upon the naive search approach by utilizing *in-situ learning* (see Appendix A.3) to exploit knowledge gained during the scope of the autotuning session. By bundling ensembles of individual techniques in a meta-technique and running them simultaneously to generate desired results, techniques that discover good configurations can be allocated larger testing budgets, while poorly performing techniques are given smaller budgets or disabled.

This was achieved in OpenTuner using the *multi-armed bandit with sliding window, area under the curve credit assignment* (AUC Bandit) meta-technique, building on previous work using a similar technique for operator selection rather than technique selection in [22].

In addition, techniques in a meta-technique can share information through a local results database, resulting in a collaborative effect that further improves search performance. As a result, an OpenTuner meta-technique provides an in-situ learning search approach whose *search affinity* (see Appendix A.5) for a representation exceeds that of the most efficient technique in the bundle. This makes in-situ learning search approaches significantly more robust to variations in representation when compared to a naive search approach.

Still, the library of techniques has been constantly growing since the creation of OpenTuner. With the

Parameter Type	Number of Operators
PermutationParameter	12
HalideComputeAtScheduleParameter	12
ScheduleParameter	12
BooleanParameter	4
PowerOfTwoParameter	6
Total Combinations	41472

Table 1.1: Number of operators for parameter types in the Halide domain
Each parameter type may take several different operators. Combining these give 41472 different choices for operator sets. The *HalideComputeAtScheduleParameter* is a custom parameter type extending the *ScheduleParameter*. See Chapter 4 for more about parameter types and operators.

introduction of *composable techniques* (see Appendix A.3) presented in this thesis, the number of potential techniques to use for a representation has exploded even further. This is because every combination of *operator* choices with a *technique base* results in a different technique.

For instance, representations in the Halide domain contain five different kinds of parameter types, as seen in Table 1.1. For each parameter type, we choose one from the multiple operators available. Combining these choices yields over 40,000 different choices for picking operators per technique base, and this number will only grow further as new operators are developed and introduced.

This far outstrips the number of techniques that can be reasonably bundled within an in-situ meta-technique. A typical tuning run may end before even 1000 configurations are tested, meaning the meta-technique will not even have time to try all of the techniques, let alone determine their relative performance.

As a result, an in-situ learning search approach may still perform poorly despite its robustness if the meta-technique happens to lack a technique with good search affinity with the representation. With so many techniques, performing an exhaustive or manual search to determine which techniques to include for every new representation is also highly impractical for a user. Trying to account for collaborative effects only exacerbates the issue, as entire sets of techniques must be considered, of which there are exponentially many in the number of techniques.

As shown in this thesis, the default in-situ learning search approach in OpenTuner is indeed no longer sufficient for some problems, creating a need for a more robust search approach capable of efficiently and intelligently exploring the vast space of techniques. So, to achieve this, just as OpenTuner used the meta-technique to take the naive search approach to the next level of in-situ learning, this thesis presents Deep-Tuner, a recommender system, to take the in-situ learning search approach to the next level of *transfer*

learning (see Appendix A.3).

1.3 DeepTuner

The problem of choosing the right techniques to use for a given representation reflects the "deep" nature of general program autotuning. Just as the original problems in OpenTuner can be cast as a search space of configurations and explored using various search approaches, at the next level, the choice of search approach could in turn be cast into a search space and explored using some other search algorithm. In essence, the autotuning process itself could be framed as an autotuning problem. Extending this notion leads to infinitely many layers in autotuner systems, each being tuned by the layer above.

Each successive layer neccessarily operates over a larger scale of data, as a single configuration in the search space of one layer is an entire tuning process consisting of the evaluation of many configurations in the layer below it. As such, in-situ knowledge from a single tuning run is no longer sufficient for operating at the next layer of autotuning to efficiently explore the space of techniques. Rather, search approaches capable of transfer learning are required.

As seen in Figure 1.1, a transfer learning search approach in DeepTuner takes the in-situ search approach of OpenTuner to the next level from a machine learning aspect. A naive search approach, only performs learning using evaluation results within the scope of a single technique. An in-situ search approach built on this by using a meta-technique and performing learning using results within the scope of the entire local autotuning session. In a transfer learning search approach, the meta-technique not only uses evaluation results from within the current local autotuning session, but also uses global results from past autotuning sessions.

In this thesis, we present DeepTuner as a proof of concept to demonstrate the power of transfer learning over traditional in-situ learning methods in OpenTuner. DeepTuner is an open source extension to OpenTuner providing a global data repository and technique recommendation service hosted on a private cloud to enable knowledge transfer across OpenTuner tuning runs.

DeepTuner leverages a *cross-session database* (see Appendix A.5) which stores historical *search affinity* (see Appendix A.5) between techniques and representations across old tuning runs. The cross-session database not only provides quick access to aggregate statistics over historical data about particular techniques or representations, but also enables DeepTuner to perform intelligent technique recommendation, even for new representations that have not been encountered. Recommendation of techniques on new representations is achieved by comparing historical search affinity data with ongoing evaluation results to determine representations that may have a similar topography. DeepTuner's recommendation endpoints then facilitate

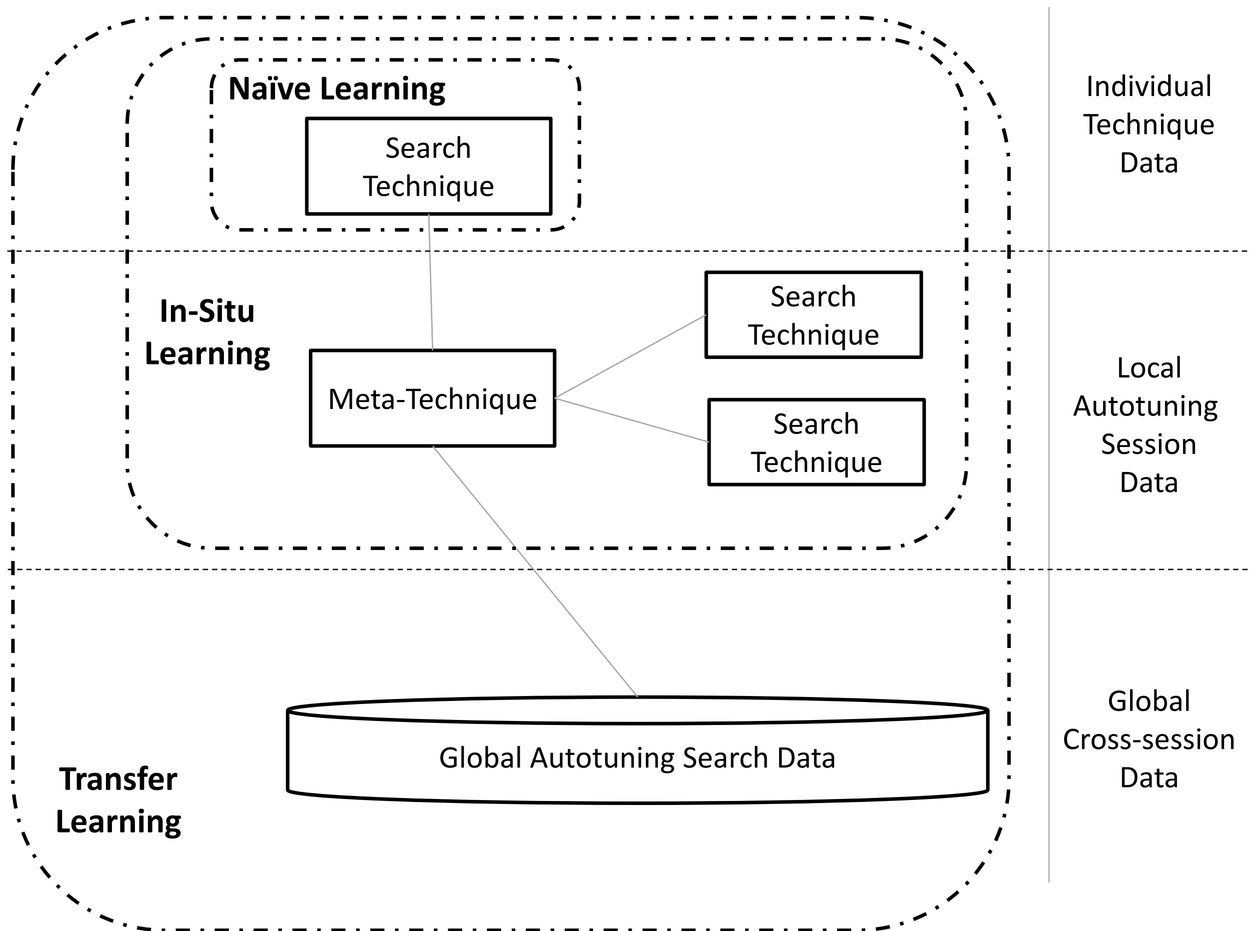


Figure 1.1: Levels of learning in OpenTuner search approaches
Deeper levels of learning exploit data from wider scopes. While naive and in-situ learning only interact with data from the local tuning run, transfer learning makes use of global data from historical tuning runs

the creation custom meta-techniques that can take advantage of transfer learning.

Thus, the recommendation layer provided by DeepTuner essentially acts as the next autotuning layer above OpenTuner. These recommendation algorithms fill the role of search algorithms exploring the space of search approaches. Recommended search approaches are "evaluated" by transmitting recommended changes to meta-techniques in OpenTuner via DeepTuner's endpoints, followed by receiving performance feedback from those meta-techniques. The received feedback informs future recommendation, allowing for further fine tuning of the transfer learning meta-technique.

The end result is a more robust search approach that can simultaneously search the space of configurations while efficiently exploring the space of techniques for a technique with good search affinity for the representation. In other words, DeepTuner essentially autotunes and improves the search approach in OpenTuner as that search approach is autotuning the user's problem.

DeepTuner greatly improves upon what OpenTuner could do. We highlight some of the different questions

Search Technique Class	Search Affinity
GreedyComposableTechnique	0.02255
PSO (particle swarm optimization)	0.01288
AUCBanditMutationTechnique	0.01288
GGA (global genetic algorithms)	0.00481
RandomThreeParentsComposableTechnique	0.004606
NormalGreedyMutation	0.00453
PatternSearch	0.00435
PseudoAnnealingSearch	0.00097
DifferentialEvolution	0.00079
UniformGreedyMutation	0.00075
GA (genetic algorithms)	0.00075
PureRandom	0.00068

Table 1.2: Aggregate search affinity for technique classes on the brazil58 TSP benchmark
The search affinity is aggregated over the historical autotuning data in the cross-session database (see Section 5.4) and obtained using one of DeepTuner's auxiliary endpoints (see Section 5.4.1).

and problems that DeepTuner can now address:

1. **What is the best technique base for my problem?**

   A developer interested in contributing to the library of techniques may often be interested in the relative search affinities of existing techniques. Because of how the cross-session database tracks search affinities for all techniques across past tuning runs, finding out this type of information is now much more straight forward. For instance, if the developer wants to create a technique for a particular domain, they may wish to know which technique base currently has the greatest search affinity.

   The user can obtain a set of search affinities for all techniques, filtered by the particular domain they are interested in. Thanks to the standardization of default technique names, the technique base of a technique can be determined using the root of the default name or through a lookup of the default library of technique names distributed with OpenTuner. Using this, the search affinities can be aggregated by the technique base to get relative search affinities of the technique bases.

   For example, Table 1.2 gives the list of technique classes sorted by their overall search affinity on the brazil58 benchmark for the travelling salesman problem autotuner. This same filtering and aggregation could be applied at different levels, such as the domain level instead of the problem level or the operator level rather than the technique base level, to answer a variety of similar questions.

2. **Is my representation good?**

   Users implementing an autotuner in OpenTuner are often interested in whether a new representation they have created for their problem is "good". More specifically, they wish to know whether there is good search affinity for techniques interacting with the new representation. The cross-session database of DeepTuner allows the user to compare the best search affinity achieved by techniques in the new representation to that of alternative representations. This gives a way to measure how easily searched the new representation is relative to some baseline, even if the parameters in the representation or objective function have changed significantly to the point where different techniques are optimal.

3. **How similar is my problem to other problems?**

   While problems within the same domain tend to be related, the degree to which they differ can vary. Using DeepTuner's cross-session database, a user can not only compare the similarity of autotuning search spaces on the basis of their parameters in the representation, but can also compare their topographies based on their interaction with various techniques. Performing statistical correlation on the relative search affinities of techniques between two problems gives a measure of similarity that better accounts for the topography of the autotuning space instead of just the dimensions of the autotuning space. Performing this for problems within a domain can sometimes reveal clustering of problems and other insights.

4. **Given a new problem, what search approach do I use?**

   Before, when faced a new representation, an OpenTuner user would often select or search for the best in-situ search approach from a very small set of pre-made search approaches. Even for old representations, this selection process was often anecdotal, with user A choosing an in-situ search approach other than the default because user B said it "worked well" for some other representation. Exploring in-situ search approaches outside of the pre-made sets by choosing a custom bundle of techniques was even less common.

   With DeepTuner, we introduce meta-techniques taking advantage of the recommendation endpoints. By utilizing the data from the current tuning run and past tuning runs, these meta-techniques make use of transfer learning and intelligently search the space of techniques to use. These new search approaches effectively act as in-situ search approaches that update the techniques in their bundles to approach a near optimum. Thus we get a more robust self-optimizing, self-updating search approach that essentially automates the old process of manually searching for the right search approach to use.

   Figure 1.2 shows the difference between the default in-situ learning search approach in OpenTuner and

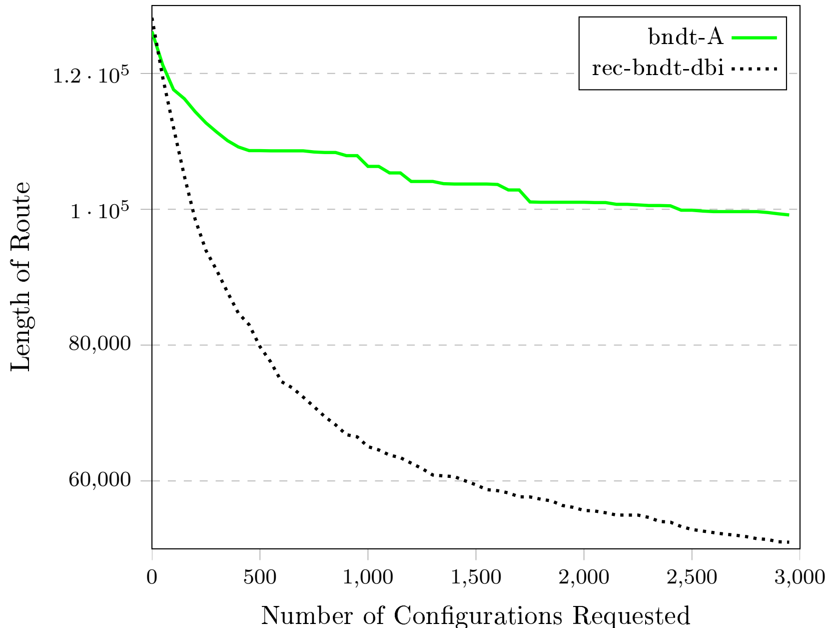


Figure 1.2: Autotuning performance of an in-situ learning vs. a transfer learning search approach
Best route length found as a function of configurations tested for the Travelling Salesman Problem att48 benchmark. Performance is shown as the median over 75 tuning runs and lower route lengths are better. The current default in-situ learning search approach in OpenTuner (bndt-A) performs much worse than a transfer learning search approach (rec-bndt-dbi).

the DBI recommendation bandit meta-technique introduced in Section 6.1 which can take advantage of transfer learning.

1.4 Contributions

We consider the work in this thesis to have the following contributions:

- While there has been work on the past on creating hybrids of different heuristic techniques, we make an attempt to formalize technique composition. This enables us to easily express a variety of new technique variants in OpenTuner.
- We introduce DeepTuner which maintains a cross-session database for historical autotuning data to provide technique recommendation on top of OpenTuner. This facilitates the creation of new meta-techniques - the recommendation bandit meta-technique and DBI recommendation bandit meta-technique - used for transfer learning search approaches.
- We demonstrate the robustness of the new search approaches by comparing the recommendation bandit meta-technique to the traditional in-situ learning AUC bandit meta-technique on nine representations

across three domains using a variety of initial technique sets. For almost all representations and initial technique sets, the recommendation bandit meta-technique achieves performance on-par with or significantly greater than that of the in-situ learning search approaches.

- We present a self-initializing and self-updating search approach using the DBI recommendation bandit meta-technique, essentially automating the old user-controlled process of selecting a good search approach. The new search approach matches the performance of the old search approaches typically selected by OpenTuner users in the PetaBricks and Halide domains, and achieves up to 3.7x improvement over the old default search approach on TSP benchmarks.

1.5 Organization

This thesis introduces several new terminologies which are defined in Appendix A. The remainder of this thesis is organized as follows:

Chapter 2 discusses related works.

Chapter 3 summarizes the original OpenTuner framework.

Chapter 4 introduces technique composition.

Chapter 5 details the cross-session database used by DeepTuner

Chapter 6 discusses search approach recommendation in DeepTuner and introduces the recommendation bandit meta-technique, which takes advantage of technique recommendation in DeepTuner.

Chapter 7 presents autotuning performance results.

Chapters 8 and 9 discusses future work and conclusions.

Chapter 2

Related Work

To the best of our knowledge, a system for formalizing technique hybridization and for technique recommendation in a general autotuning framework has not been attempted before. Instead, we discuss related work on the major components that DeepTuner brings together: autotuning frameworks, technique hybridization, and algorithm recommendation.

2.1 Autotuning Frameworks

While several program autotuning systems and frameworks exist outside of OpenTuner, they are often not as general in scope and aimed towards specific domains.

2.1.1 Linear Algebra

ATLAS is an autotuning system aimed towards linear algebra kernels, which produces a highly optimized matrix multiply for a given architecture [32]. This is achieved by exhaustive search where a variety of generated alternate implementations are empirically evaluated.

In a similar vein, OSKI providing a library of several self-tuning sparse matrix kernels including sparse matrix-vector multiply and sparse triangular solve [30]. The kernels are are tuned at runtime through a combination of exhaustive search and heuristics to select the fastest implementation in OSKI for the given machine and matrix.

Additional work has been done by Choi et al. on an autotuning framework specifically for sparse matrix-vector multiply on GPUs using performance models to guide the autotuning process [6].

2.1.2 Signal Processing

The SPIRAL system provides a platform-tuned library of linear signal processing transform algorithms [24]. SPIRAL achieves this by combining formula generation of alternate high-level algorithms for a particular signal processing transform with code generation of possible implementations for a chosen formula to obtain a space of possible alternate implementations. This space is searched using a combination of naive search approaches including exhaustive search, dynamic programming, random search, a hill-climber, and a stochastic evolutionary algorithm.

Within this space, the FFTW library utilizes autotuning to yield machine-optimized discrete Fourier transform computations [13]. FFTW works by recursively decomposing a DFT problem - defined by the shape of the input data - into smaller sub-problems. This continues until sub-problems are sufficiently small, at which point FFTW selects from a variety of alternate optimized codelets that directly solve the sub-problem. The space of possible decompositions is explored using a combination of exhaustive search and dynamic programming.

2.1.3 Stencil Computations

Program autotuning has also seen very successful application to the domain of stencil-based computation - kernels operating over a possibly multi-dimensional array in a fixed manner. A general autotuning framework for transforming serial stencil expressions in Fortran 95 into tuned parallel implementations in Fortran, C, or CUDA can show performance gains of up to 22x speedup [17].

Rather than using explicit code, the autotuning framework PATUS generates stencil computation kernels for multicore CPU and GPU from a stencil specification using a C-like syntax [7]. Alternate implementations are generated according to a strategy template outlining how the kernel should be optimized and parallelized. The strategy can be either chosen from a pre-defined set or custom-designed, and the final set of alternates is explored by a user-specified naive search approach such as multi-run Powell search, Nelder-Mead, or an evolutionary algorithm.

Sepya, a domain-specific language for stencils embedded in Python, utilizes an autotuning compiler to output high-performance stencil code [18]. User-defined stencils in Sepya are transformed using a proved set of optimizations to target multi-core CPUs specifically, and the best implementation is determined via exhaustive search.

Meanwhile, PARTANS provides an autotuning framework specifically for distributing a stencil computation kernel across multiple GPUs [20].

2.1.4 Runtime Systems

The Active Harmony framework is aimed towards autotuning of runtime applications, which is enabled by infrastructure for exposing multiple variants of underlying libraries [28]. The space of possible combinations of variants is searched using a Nelder-Mead simplex method. Active Harmony was combined with CHiLL, a compiler framework for automatically generating alternate implementations of code, to achieve autotuning of compiler-generated code [29].

2.2 Search Technique Hybridization

Our decision to attempt to formalize technique composition in OpenTuner stems from the success that hybrid techniques have shown in various domains.

For example, a hybrid search algorithm combining a genetic algorithm (GA) with simulated annealing has seen successful application in flow-line scheduling, with the hybrid algorithm outperforming both of the original algorithms [23]. A GA and simulated annealing hybrid has also been applied to flow shop scheduling [21].

The emerging class of memetic algorithms also exhibit technique hybridization. Memetic algorithms extend population-based global search algorithms such as evolutionary algorithms with methods for local optimization. For instance, memetic algorithms combining an evolutionary algorithm with a hill-climbing crossover have been applied to a wide range of problems [19],[31].

Genetic algorithms have also been combined with particle swarm optimization (PSO) by incorporating swarm behaviour into the genetic algorithm. Work has been done showing that a hybrid of GA and PSO outperforms both GA and PSO in recurrent network design problems [16]. Similar success was also demonstrated for a PSO-GA hybrid in the context of optimizing several benchmark non-linear functions [26]. The hybrid showed faster convergence and a higher ability to find the global optimum over both PSO and GA, .

Hybridization of PSO and GA has also been attempted in the other direction using PSO as the base. This was achieved by incorporating GA operators into the velocity and update rules of PSO and the resulting hybrid was again capable of outperforming both PSO and GA on quality-of-service multiast routing problems [3]. In addition, PSO has been hybridized using a differential evolution operator, which showed similar positive results [34].

2.3 Algorithm Recommendation

Recommendation systems for algorithmic approaches have been successfully applied in several other domains besides search approach recommendation in program autotuning.

One such system is Delphi, a machine learning as a service system that both automates and optimizes the classifier selection process [10]. Given a dataset and performance metric by the user, Delphi recommends a fully trained and parametrized classifier that best satisfies that performance metric. This entails selection of a type of classifier algorithm, such as a support vector machine using a radial basis function kernel or a decision tree with entropy cuts, along with any parameters needed for the algorithm. This parallels our process of recommending fully specified techniques that best search a representation given by the user.

Work has also been done by Stephenson on collaborative compilation [27]. In collaborative compilation, clients opt to insert low-overhead instrumentation code to measure the effectiveness of different compiler optimizations on their given application and data set. The cross-compilation instrumentation data is aggregated in a global knowledge base, and is then used to perform transfer learning on what compiler heuristics and optimizations work best. The best predicted optimizations are then recommended back to clients as a policy for meta-optimizing the compiler algorithm.

Chapter 3

The OpenTuner Framework

OpenTuner treats autotuning as a search problem. Each point in the search space is a *configuration* - a concrete assignment of values to a set of parameters. See Section 4.2.1 for more on parameter types available. We call this set of parameters that defines the dimensions of the search space the *parameters in the representation*. A *result* is the evaluation of a configuration according to a user defined *objective function*. The combination of the parameters in the representation and the objective function specify the topography of the search space. These are specified for an autotuner implemented in OpenTuner when a user creates a *problem definition*.

The search space is then explored by a user-specified *search approach*, which makes requests for configurations to be evaluated. These requests are called *desired results*. This interaction between the search approach and the problem definition makes up the search aspect of autotuning in OpenTuner.

Actual measurement of desired results is handled separately by a minimal measurement driver, which is a thin wrapper around the user defined objective function. This abstraction allows for parallelism in measurement. It also separates the details of obtaining measurements for configurations from the search process which we focus on in the rest of the thesis. The results from the measurement process are communicated through a *results database*, which tracks results collected during a tuning run along with metadata about the problem definition and search approach for the tuning run.

3.1 Problem Definition

Because of their generic nature, autotuning frameworks like OpenTuner require some specification by users to construct an autotuner for their particular application. We refer to this process of defining the autotuning problem as the creation of a *problem definition*. Conceptually, a problem definition has a hierarchical

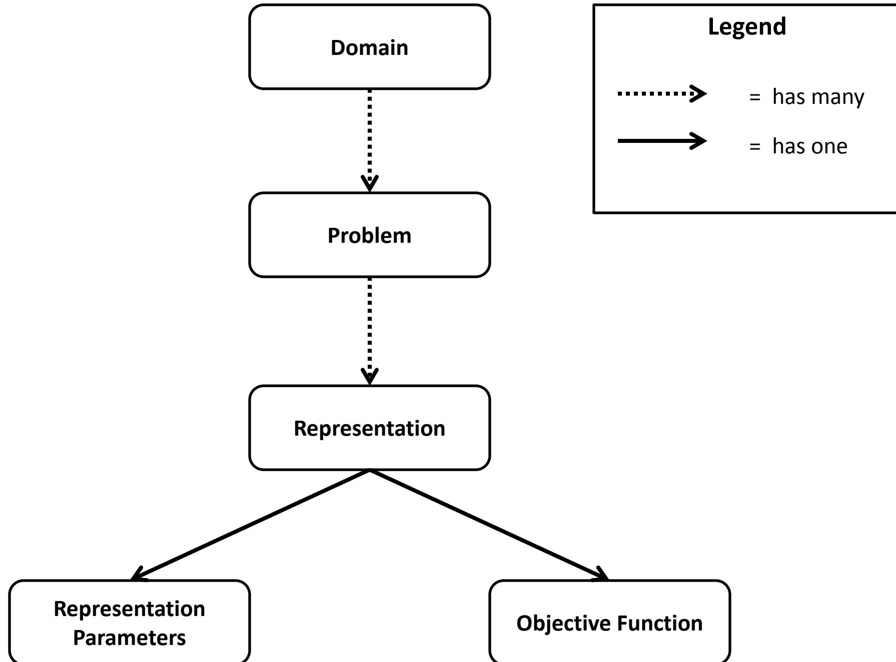


Figure 3.1: Structure of a problem definition
An autotuner implemented in OpenTuner applies over a single domain. Within this domain, the user may have many different problems that require optimization. Each can be mapped into multiple representations understood by OpenTuner. A representation consists of a set of parameters in the representation and an objective function, which OpenTuner uses as the dimensions and evaluation function for the autotuning space.

structure as seen in Figure 3.1, which consisting of a *domain* containing multiple *problems*, each of which has one or more *representations*.

For example, in the Mario autotuner mentioned in [5] , the domain would be beating a level in Super Mario. The problem would be the particular level we are trying to beat (e.g. World 1-1). As for the representation, we could use a naive representation consisting of many booleans, each corresponding to whether a particular input button is being pressed at a particular frame. Or, we could use a representation encapsulating the idea of durations, with several enumeration parameters choosing movement direction paired with integer parameters specifying the duration of the movement. This is decoupled from jumping, which is encoded with integer parameters specifying which frames to jump on and the duration that each jump should be held.

3.2 Search Approach

After creating a problem definition, the user can then proceed to autotune a particular problem using one of the defined representations. At the start of the autotuning session or *tuning run*, the user specifies the name of a search approach from a registry of named, instantiated search approaches. The search approach decides how the search driver explores the autotuning space by controlling which desired results the limited testing resources of the measurement driver will be spent on. The desired results are generated by using various techniques in OpenTuner's extendible library of techniques. Each *technique* can be either an implementation of known search algorithm - such as differential evolution, Nelder-Mead or Torczon hillclimbers, pattern search, or particle swarm optimization - or could be a generated composable technique. See Chapter 4 for more detail on technique composition in OpenTuner.

An individual technique can be instantiated as a naive search approach. However, these are often unable to handle the diverse range of search spaces. In order to provide more robust search approaches that take advantage of in-situ learning, Opentuner uses *meta-techniques* - collections of individual techniques. A meta-technique acts as an ensemble of techniques where the meta-technique decides which techniques gets to submit the next desired result. Even the simplest meta-technique, such as a round robin meta-technique which cycles through techniques in order, can take advantage of in-situ learning as results from all techniques are shared through the results database. Thus, individual techniques are able to learn about and benefit from good configurations discovered by other techniques in the ensemble. This collaborative effect boosts the performance of the search approach as a whole.

3.2.1 AUC Bandit Meta-Technique

The core meta-technique used for in-situ learning in OpenTuner is the *multi-armed bandit with a sliding window, area under the curve credit assignment* (AUC Bandit) meta-technique. This extensible meta-technique is based on an optimal solution to the multi-armed bandit problem presented by Fialho et al. in the slightly different context of operator selection [11]. In the multi-armed bandit problem, there are multiple levers on a slot machine, each with an unknown probability of paying out. The goal is to pick lever-arms in a way to maximize the expected pay out. In the context of OpenTuner, the levers correspond to the techniques and their payouts correspond to autotuning performance of the technique. The multi-armed bandit problem encapsulates the problem of trading off between *exploitation* (using the best known technique) and *exploration* (estimating the performance of each technique).

The AUC Bandit meta-technique assigns each technique, t, a score based on a sliding window of recent history and assigns testing resources to the technique with the highest score. The score is calculated according

to the formula

$$AUC_t + C\sqrt{\frac{2\log H}{H_t}}$$

where AUC_t is the credit assignment term quantifying performance of t, C is the constant determining the exploitation-exploration trade-off, H is the length of the sliding window, and H_t is the number of times t has been used within the sliding window of history. AUC_t acts as the exploitation term that increases with better technique performance within the history window, while the second term acts as the exploration term and decreases as t is used more by the meta-technique. Values for C and H are hyper-parameters for the AUC Bandit meta-technique and are fixed upon instantiation.

AUC_t is calculated according to the area under the curve assignment mechanism based on Fialho et al.'s work [12]. For each t, we draw a curve by based on the history of results for t within the sliding window and whether or not each result was a new best. Traversing the history for t chronologically, each new best increases the height of the curve, while results that were not new bests do not change the height. Calculating the area under the curve, scaled to a maximum value of 1 if all results were new bests, gives AUC_t. In other words,

$$AUC_t = \frac{2}{H_t(H_t+1)}\sum_{i=1}^{H_t} iV_{t,i}$$

where $V_{t,i}$ is an indicator variable with value 1 if the ith use of t in the sliding history window resulted in a new best and with value 0 otherwise.

By weighing exploration against exploitation, the AUC Bandit meta-technique effectively performs an extra layer of in-situ learning during the tuning run to dynamically favor better performing techniques, resulting in more robust autotuning performance.

Chapter 4

Search Technique Composition

Ultimately, search approaches in OpenTuner rely on the efficacy of the individual techniques. Many recent works have shown the effectiveness of composite techniques - for instance insertion of a PSO enhancement step for elites in an evolutionary algorithm, or replacement of standard crossover operator of an evolutionary algorithm with a hill-climbing operator [16],[19]. However, we recognize that writing an entirely new technique from scratch for each potential composition is inefficient and not scalable. In this work, we make an attempt to formalize the process of creating composite techniques through the idea of composable techniques, which allow for easy substitution of newly developed operators, such as a new crossover for permutations, to generate new techniques.

We break techniques into three primary stages when generating configurations to test:

1. **Selection of parent configurations**

   Conceptually, all configurations generated by a technique are derived from some original set of configurations. This set could be a single random configuration, or a subset of a population satisfying a some criteria (like having the highest fitness scores).

2. **Generation of a child configuration**

   After getting a set of parent configurations, a child configuration is generated as a testing candidate by combining the parameter values of the parents in some fashion, whether it's through crossovers or mutations, or something else altogether.

   Because combining values of different types of parameters is impractical, the generation of a child can be further partitioned into a set of independent operations over each parameter. We refer to these operations that combine parameter values as *operators*.

3. **Update of technique state**

   Finally, the fitness of the generated child is used to update the technique state to inform behaviour in following iterations. This state is often tracked through some population.

   For example, the technique could choose to replace a random population member with a lower fitness than the child configuration with the child configuration.

In traditional techniques for search and optimization, these stages - especially the selection of parents and the methodology for generating a child - were not considered independently. However, we note that they are in fact completely independent. So, we choose to separate a technique into two major components: the set of operators controlling the primary aspects of child generation and a *technique base* defining the parent selection and update processes.

By modularizing a technique this way, the technique base becomes the base for technique composition. On top of this, the choice of operators can be interchanged to generate a wide range of composite techniques. This style of technique composition offers several advantages for developing new techniques.

First, by abstracting out operators, developers can now focus their efforts on particular aspects of a technique. For instance, one could choose to solely research new operators for *PermutationParameters* to improve search performance over representations with permutations. Or, one could focus only on exploring variations in population reproduction for evolutionary algorithms to develop new technique bases.

Secondly, technique composition aids in the integration of new parameter types and operators. Previously, the introduction of new parameter types or operators required overhauling existing techniques to take advantage of them. However, this often required intimate knowledge of old techniques that the developer of the operator lacked. With composable techniques, the new operators or parameter types can simply be swapped into the operator map, without knowledge of how the underlying technique base operates.

Finally, composable techniques greatly reduce the amount of code needed to express a much richer set of possible technique base - operator choice pairings, as the number of possible operator choices grows combinatorially with number of parameter types and operators per parameter type.

4.1 Creating a Technique Base

To make the development of composable techniques capable of technique generation easier, we present the *ComposableEvolutionaryTechnique* class as an extendible class for specifying new technique bases. Implementing a new technique base only requires defining the following methods corresponding to the three primary stages:

- **get_parents(population)**

  Given the current population, this method returns a list of configurations to use as parents. One of these is specified as the primary parent.

- **select_parameters(params)**

  Given the set of all parameters in a configuration, this method returns a subset of parameters to apply operators to when combining configurations. This controls the extent to which parent configurations are combined.

- **update_population(config, population)**

  Given a configuration that has been evaluated and the original population, this method should update the state of the population.

An initialized technique base generates configurations during a tuning run by first initializing a population of configurations, then calling the above methods in succession to generate new configurations. The configurations are generated from the parents by applying a set of operators specified by an *operator map*. If there is no operator specified for a particular parameter type, then a default operator is used. This is the "no-op" operator unless otherwise overridden by the creator of the technique base.

The population consists of instances of a *PopulationMember* class, which is a thin wrapper around a configuration. This class can be extended by a technique base developer to provide custom functionality if needed. The size of the population is specified as a hyper-parameter to the technique base which defaults to 30.

Currently, the OpenTuner's technique library contains two technique bases based on existing techniques. Their behaviours are summarized in Table 4.1.

4.2 Choosing a Set of Operators

The set of operators to use with a technique base is specified during instantiation by the operator map. An operator map is a dictionary from OpenTuner parameter types to the name of an operator method associated with that parameter type. The operator map also tracks any user specified operator hyper-parameter - value pairs, such as the crossover length in a permutation operator, as well as a list of any additional arguments. This data structure is initialized or can be randomly generated when instantiating a technique base and allows for easy creation of hybrid techniques.

GreedyComposableTechnique	
get_parents(population)	Cycles through each population member, selecting that population member along with the global best configuration as parents. The population member is specified as the primary parent.
select_parameters(params)	Each parameter has a random probability to be chosen. The probability is controlled by a crossover rate hyper-parameter to the technique base.
update_population(config, population)	If the child configuration is better than the original population member, then the original population member is replaced by the new configuration
RandomThreeParentsComposableTechnique	
get_parents(population)	Three parents are randomly selected from the configuration. There is a fixed probability of including the global best configuration as one of the parents. This probability is controlled by a hyper-parameter to the technique base.
select_parameters(params)	Each parameter has a random probability to be chosen. The probability is controlled by a crossover rate hyper-parameter to the technique base.
update_population(config, population)	The child configuration replaces the oldest member of the population.

Table 4.1: Implemented technique bases in OpenTuner

The *GreedyComposableTechnique* is based on the many greedy search techniquesin OpenTuner. The *RandomThreeParentsComposableTechnique* is based off of an implementation of differential evolution [2].

4.2.1 Parameter Types

OpenTuner supports a hierarchy of built-in parameter types for representing tuning variables in problems, as seen in Figure 4.1. This hierarchy can be extended by users as needed to make custom parameters. A parameter inherits properties of its superclass, including any operators.

The parameters can be split into two main groups: *PrimitiveParameters* (also called *NumericParameters*) and *ComplexParameters*. These and their subtypes are described in Table 4.2.

Parameter Type	**Description**
NumericParameter	a numeric value with an upper and lower bound
IntegerParameter	an integer value with a legal range
FloatParameter	a continuous float value with a legal range
ScaledNumericParameter	a value searched on a separate scale than its concrete value
LogIntegerParameter	an integer value searched on a log scale
LogFloatParameter	a float value searched on a log scale
PowerOfTwoParameter	takes on values that are powers of two. Searched on a log scale
ComplexParameter	Domain specific structures that lack well defined gradient functions
BooleanParameter	a boolean value that is either true or false
SwitchParameter	an unordered collection of choices represented as integers
EnumParameter	an unordered collection of choices taken from an arbitrarily typed list
Array	a collection of related values of the same type
BooleanArray	an array of boolean values
FloatArray	an array of float values
PermutationParameter	an ordering of items
ScheduleParameter	a permutation with a set of dependencies that restrict legal orderings
SelectorParameter	a permutation where each indexed location is associated with an integer. Contains a *PermutationParameter* and several *LogIntegerParameters* as sub-parameters

Table 4.2: Descriptions of parameter types in OpenTuner

4.2.2 Available Operators

Each parameter has a set of operators. These operators take in a set of parent configurations and mutate the corresponding parameter value in the first configuration according to the parent values. The operators form the set of available transformations for generating new configurations.

OpenTuner standardizes naming for operator methods, which can be identified by the prefix 'op#_', where # is the number of required input configurations. The prefix 'opn_' specifies an arbitrary number

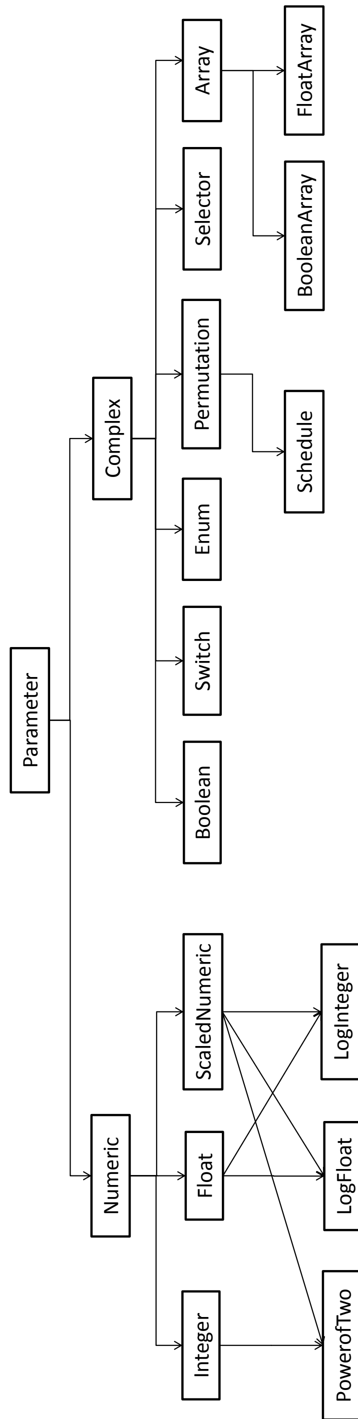


Figure 4.1: Hierarchy of OpenTuner's built-in parameter types
OpenTuner comes with several parameter types commonly used in autotuners. The parameter types can be extended to handle custom domain-specific data structures.

of inputs as a list. The first argument into an operator is always the configuration that will be mutated. This is followed by the required parent configurations, then any required arguments, and finally optional arguments.

To prevent passing insufficient parent configurations into a operator, all technique bases are defined with a value for the minimum number of parents guaranteed by the *get_parents(population)* function. A technique base is prevented from using any operator requiring more input configurations than it can guarantee.

Tables 4.3, 4.4, and 4.5 list out and briefly describe the available operators for each parameter type. Note that operators are inherited, so we do not re-list operators for sub-classes unless the behaviour changes. In the descriptions, the parameter value refers to the value of the parameter for the configuration being mutated (i.e. the child configuration).

Parameter	
op1_noop	the null operator. Does nothing.
op1_randomize	sets the parameter's value to a random value
op3_swarm	stochastically moves the parameter's value towards the values in the other two parent configurations. This attempts to simulates the update step in particle swarm optimization for the general case.
op4_set_linear	sets the parameter value to a linear combination of 3 other configurations. Takes in weights for the parent configurations as additional arguments.
opn_stochastic_mix	stochastically recombines the list of parent values to obtain the final parameter value. This is done by randomly taking the value of parent according to a list of weights given as an additional argument.
NumericParameter	
op1_scale	scales the parameter's value by a constant factor given as an additional argument.
op3_difference	sets the parameter's value to the difference between two parent values.
opn_sum	sets the parameter's value to the sum of the parent values.
IntegerParameter	
op3_swarm	performs a single update step in particle swarm optimization for integers, returning the new velocity $$c * velocity + r_1 * c_1 * (cfg_1 - cfg) + r_2 * c_2 * (cfg_2 - cfg)$$ where cfg is the original parameter value; cfg_1 and cfg_2 are the parent values; c, c_1, and c_2 are hyper-parameter weights; *velocity* is an additional argument representing the old swarm velocity; and r_0 and r_1 are random values between 0 and 1. The parameters's value is set to the new velocity with gaussian noise added. The standard deviation of the gaussian is an additional hyper-parameter
FloatParameter	
op3_swarm	performs a single update step in particle swarm optimization for floats, returning the new velocity $$c * velocity + r_1 * c_1 * (cfg_1 - cfg) + r_2 * c_2 * (cfg_2 - cfg)$$ where the values are as in the operator for *IntegerParameters*. The parameters's value is set to the new velocity.

Table 4.3: Available operators in OpenTuner by parameter type

ComplexParameter	
op4_set_linear	attempts to simulate setting the parameter's value to a linear combination of the parents for common cases, though not possible in general.
BooleanParameter	
op1_flip	flips the parameter's value.
op3_swarm	simulates a single update step in particle swarm optimization by updating the current position and returning the new velocity $$c * velocity + r_1 * c_1 * (cfg_1 - cfg) + r_2 * c_2 * (cfg_2 - cfg)$$ where the values are as in the operator for *IntegerParameters*. The boolean value True is taken as a value of 1, and False is taken as a 0. The parameters's value is set probabilistically according to the new velocity.
Array	
op3_cross	crosses two parent arrays by replacing a random subsection of the first with the corresponding subsection of second.The relative size of the subsection is controlled by a hyper-parameter.
op3_swarm	simulates the iterative step in particle swarm optimization for arrays. Probabilistically moves towards one of the parent values by performing a crossover with the parent value. The weights for the parent and child configurations as well as the strength of the crossover are set by hyper-parameters.
BooleanArray	
op3_swarm_parallel	performs the iterative step of particle swarm optimization at each individual location in the array according to the same formula as with the *BooleanParameter* $$c * velocity + r_1 * c_1 * (cfg_1 - cfg) + r_2 * c_2 * (cfg_2 - cfg)$$ The weightings c, c_1, and c_2 are again hyper-parameters. r_1 and r_2 are random values from 0 to 1, however the same random values are used across all locations.
FloatArray	
op3_swarm_parallel	performs the iterative step of particle swarm optimization at each individual location in the array according to the same formula as with the *FloatParameter* $$c * velocity + r_1 * c_1 * (cfg_1 - cfg) + r_2 * c_2 * (cfg_2 - cfg)$$ The weightings c, c_1, and c_2 are again hyper-parameters. r_1 and r_2 are random values from 0 to 1, however the same random values are used across all locations.

Table 4.4: Available operators in OpenTuner by parameter type (continued)

PermutationParameter	
op1_small_random_change	iterates through the parameter's list and probabilistically swaps each element with the next element. Takes the probability of swapping as a hyper-parameter.
op2_random_swap	sets the parameter's value to the parent permutation with a random pair of elements swapped.
op2_random_invert	sets the parameter's value to the parent permutation with a random subsection reversed. The size of the reversed subsection is controlled by a hyper-parameter.
op3_cross	applies a crossover operator to two parent permutations and sets the parameter's value to the resulting permutation. Which crossover operator to use and the strength of the crossover is specified by a hyper-parameter.
op3_cross_PX	based on a partition crossover by Whitley [33]. Chooses a random cut point and reorders elements in the first parent up to the cut point according to their order in the second parent.
op3_cross_PMX	based on the partially-mapped crossover by Goldberg and Lingle[14]. Replaces a random section of the first parent with the corresponding section in second. Displaced elements in the first parent are moved to the old position of the elements displacing them. The size of the replaced section is controlled by a hyper-parameter.
op3_cross_CX	an implementation of a cyclic crossover. Repeatedly replaces elements of in the first parent with the element at the same index in the second. This is done until a cycle is reached and the original permutation is valid agan. The initial replacement index is random.
op3_cross_OX1	based on an ordered crossover by Davis [8]. Exchanges a subpath from the second parent into the first while maintaining the order of the remaining elements in the first parent. The size of the exchanged section is controlled by a hyper-parameter.
op3_cross_OX3	based on a variation of ordered crossover by Deep [9]. Similar to op3_cross_OX1, except the parents have different cut-points for their subpaths. The size of the exchanged subpaths is controlled by a hyper-parameter.
op3_swarm	simulates the iterative step in particle swarm optimization for permutations. Probabilistically moves towards one of the parent values by performing a crossover with the parent value. Which crossover to use, the weights for the parent and child configurations, and the strength of the crossover are set by hyper-parameters.

Table 4.5: Available operators in OpenTuner by parameter type (continued)

Chapter 5

The Cross-session Database

A basic prerequisite for transfer learning is having access to a wealth of knowledge outside of the current tuning run in the first place. While the results database used by OpenTuner for inter-technique communication can store history for other tuning runs run locally, the amount of historical data available would be insufficient for the transfer learning.

In order to better support transfer learning, DeepTuner maintains a cross-session database capable of tracking global data across all tuning runs, along with meta-information about the problem definition and search approaches used in those tuning runs. The following chapter describes the cross-session database. Chapter 6 discusses how DeepTuner uses the cross-session database for search approach recommendation.

5.1 Database Schema

Figure 5.1 shows the relationships between entities in our database.

The basic entity in the cross-session database is the *TuningRun*, which is identified by a UUID and has a timestamp for the start-date. A *TuningRun* has a one-to-many relationship with *MetaTechniquePerformance* and their associated *TechniquePerformance* entities which track performance data at various stages of the tuning run, such as the number of new bests found by a particular technique or the total number of configurations tested so far. Each of the performance entities is linked to the associated technique or meta-technique, along with the number of seconds into the tuning run and the number of configurations tested so far. Using this data, we are able to calculate search affinity of various techniques and meta-techniques during tuning runs.

Metadata about the search approach used for a tuning run is recorded by connecting the *TuningRun* directly to a *MetaTechnique* entity, and to several *Technique* entities via an associative table. More specifics

on how search approach data is encoded is provided in Section 5.2.

Each *TuningRun* is associated with data about the problem definition via a relationship with a *Representation*. The *Representation* is connected to a *Program* which has fields for the project name, program name and version, and objective, corresponding to the domain, problem, and objective function respectively. The *Representation* corresponds to the representation and tracks the parameters in the representation, along with a field for a human-readable name provided by the creator of the representation.

The cross-session database also features auxiliary entities used to store precomputed metrics aggregated over multiple tuning runs in order to reduce the computation needed for various recommendation endpoints in DeepTuner.

5.1.1 Storing Representation Parameter Structure

Because parameters - especially custom ones - may contain sub-parameters, the parameters in the representation used for a problem can have a hierarchical structure. In addition, the number and types of the parameters are not fixed due to users being able to create custom parameter types. Combining these factors, data about the parameters in the representation used in a representation does not conform well to a traditional relational database structure. Instead, we store representation data on a *Representation* by transforming the tree structure of the parameters in the representation into a canonical plain-text format that retains all structural information. This string format is stored under the parameter_info field.

The text encoding is a list of tuples of parameter-info and counts. Each parameter-info is itself a list consisting of the name of the parameter, followed by tuples of parameter-info and counts for any sub-parameters. This recursive definition is able to capture the hierarchical structure of the parameters in the representation. Because the ordering of the parameters does not matter in a given level of the parameters in the representation' structure, we are able to save space by counting number of each type of parameter rather than listing them out. Within each level of structure, we order parameter-info tuples lexicographically. By imposing a canonical order, we ensure that redundant parameters in the representation end up with the same text encoding.

From the text encoding, we can easily decode the original tree structure of the parameters in the representation and obtain various metrics like total parameter type counts through simple tree traversal.

5.2 Encoding Search Approach Data

In order to fully describe a search approach, the cross-session database must track not only the kind of technique or meta-technique is used, but also the details of the particular technique instance.This includes

hyper-parameter values, or the operators used in the case of a composable technique.

However, the number of hyper-parameters in a technique is not fixed across techniques, nor is the datatype of the hyper-parameter. In addition, the extensible nature of OpenTuner means that the cross-session database must be able to handle completely new operators or techniques. As before, this kind of unstructured data is difficult to place into a traditional relational database format. Instead, we encode all of this information into a string as the name of the instantiated technique.

To support this, we implement default name generation for techniques. This default name is assigned to a technique when it is instantiated locally or on registration to the library of techniques. Each *Technique* entity then has a root-name - usually the class name for the technique or technique base - along with the full default name containing the root-name along with hyper-parameter and operator data.

5.2.1 Tracking Hyper-parameter Information

Each technique contains a list of hyper-parameter names corresponding to the keyword arguments used to set the hyper-parameters. The hyper-parameter data is used to generate a base-name for the technique composed of the root-name followed by hyper-parameter name, hyper-parameter value pairs delimited by commas and semicolons. The format for the base-name is as follows:

$$root; hparam_1, v_1; hparam_2, v_2; \ldots; hparam_n, v_n$$

where $root$ is the root-name, $hparam_i$ is the name of the ith hyper-parameter for the technique, and v_i is the value of the ith hyper-parameter.

5.2.2 Operator Maps

For composable techniques, the generated name must also encode the operator map, which must capture information about which operator to use for various parameter types, along with any additional arguments or hyper-parameter values needed by the operators.

Each entry for a parameter type in the operator map is encoded in the form:

$$ptype; op; [arg_1, \ldots, arg_n, [[kwarg_1, v_1], \ldots, [kwarg_m, v_m]]]$$

where *ptype* is the class name of the parameter type, *op* is the name of the operator to apply for this parameter type, and the rest is an ordered list of additional arguments, where the last element is a list of keyword-argument pairs corresponding to the operator hyper-parameters.

The final generated name consists of the technique base base-name followed by the encoded entries in the operator map separated by spaces.

5.3 Uploading Tuning Data

Tuning run data can be uploaded by a user to the cross-session database. The upload process processes a results database to extract tuning run data at various stages by "replaying" the order in which techniques submit desired results or discover new bests using local timestamps. Duplicate tuning runs in the cross-session database are prevented by reusing locally generated UUIDs as the UUIDs for *TuningRun* entities.

5.4 Historical Tuning Run Data Generation

The historical tuning data we use in subsequent chapters for transfer learning in DeepTuner was generated on Lanka, a 24-node Intel Xeon E5-2695 v2 @ 2.40GHz Infiniband cluster with 12 cores and 128GB of memory per machine. Autotuning data was collected over benchmarks for the default autotuners distributed with OpenTuner. The data covered a total of 34 different representations over 6 autotuner domain consisting of: 2 different representations for a problem in Mario, 1 GCC/G++ flags autotuner benchmark, 2 representations for Rosenbrock function optimization, 8 benchmark problems in Petabricks, 6 Halide benchmarks, and 14 benchmarks in the Traveling Salesman Problem (TSP - 8 of which were symmetric and 6 of which were asymmetric (ATSP). The number of tuning runs uploaded to the cross-session database for each representation is shown in Table 5.1

Each tuning run was performed using a randomly generated search approach consisting of an AUC bandit meta-technique with randomized hyper-parameters and a random set of 5 techniques. The history window size of the AUC bandit meta-technique was chosen from the set of values $500, 200, 100$, while the exploitation-exploration term value, c, was chosen from $0.05, 0.15, 0.45$ where higher values of c favor exploration.

The techniques have a random chance of either being chosen from the default library of techniques in OpenTuner randomly, or being generated from a registry of technique bases randomly. Generation of a random composable technique occurs by selecting a technique base followed by randomly selecting a operator to use for each parameter type in the representation.

5.4.1 Using the Cross-session Database

In addition to using the cross-session database for technique recommendation, DeepTuner provides auxiliary endpoints for extracting information in the cross-session database about a particular representation. For

Domain	Representation Name	# of Tuning Runs
Mario	naive representation	99
Mario	duration representation	299
GCC	matrix multiply	84
Rosenbrock	2D Rosenbrock function	389
Rosenbrock	4D Rosenbrock function	51
Petabricks	convolution	811
Petabricks	sort	817
Petabricks	Strassen	992
Petabricks	kmeans	899
Petabricks	tridiagonal	896
Petabricks	Poisson	797
Petabricks	matrix approximation	578
Petabricks	Helmholtz	771
Halide	wavelet	492
Halide	interpolate-simple	346
Halide	blur	482
Halide	bilateral grid	468
Halide	interpolate	517
Halide	interpolate-simplest	96
TSP	att48	432
TSP	brazil58	432
TSP	brg180	432
TSP	dantzig42	432
TSP	gr120	432
TSP	gr48	432
TSP	si175	432
TSP	swiss75	432
ATSP	br17	437
ATSP	ftv170	437
ATSP	ftv44	437
ATSP	p43	436
ATSP	rgb323	436
ATSP	ry48p	339

Table 5.1: Tuning run data contained in the cross-session database by representation

example, given a particular representation we can determine which techniques have the best known search affinity. Or we can query the particular representation's similarity to other representations according to the metrics mentioned in Section 6.1.

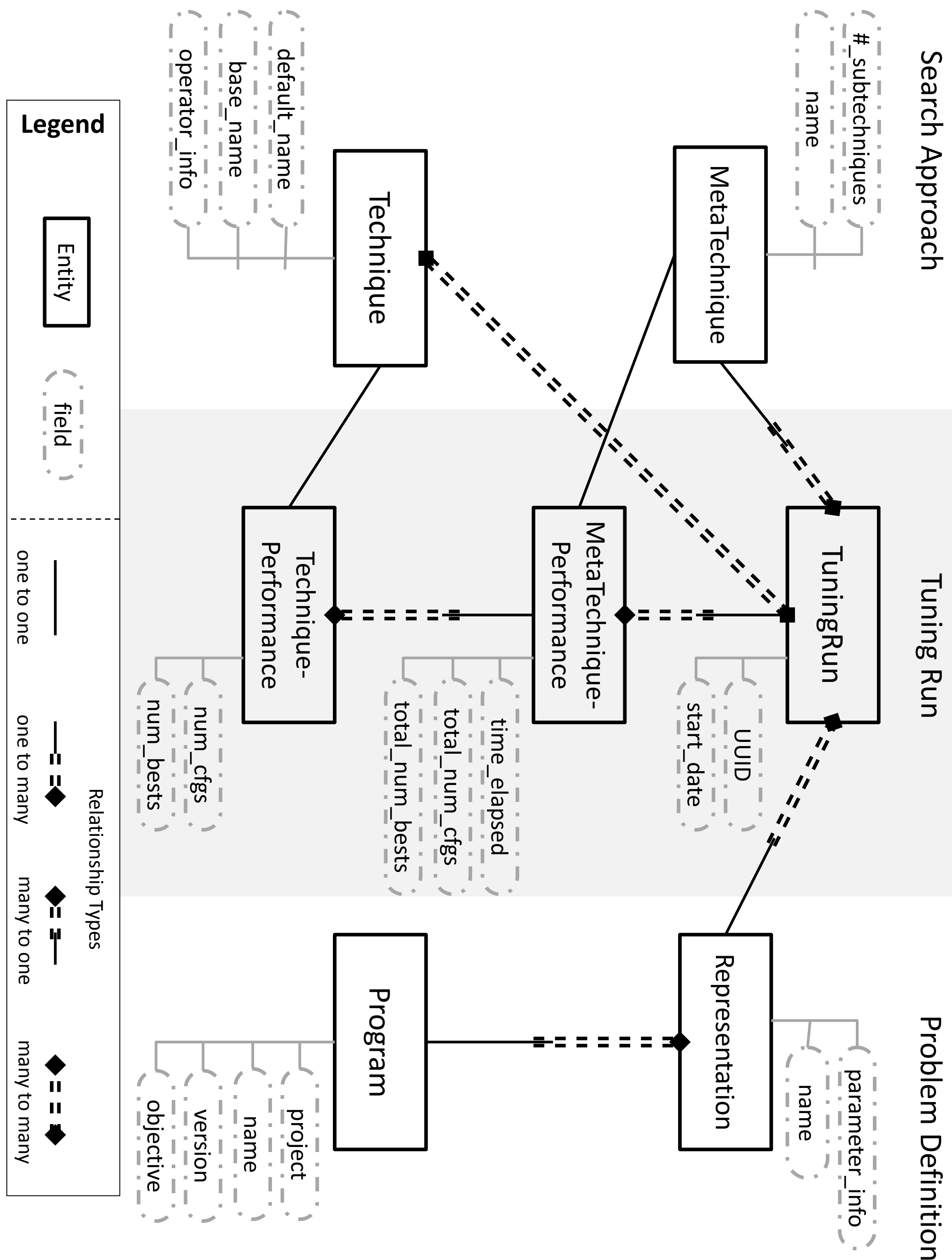


Figure 5.1: Database schema for the cross-session database
Entities can be split into 3 groups based on what they store data about: the search approach, the tuning run data, and the problem definition.

Chapter 6

A Recommender System

At a high level, the goal of DeepTuner is to provide a service for transferring knowledge from previous autotuning sessions to search approaches in OpenTuner.[1] Currently, this knowledge transfer is anecdotal, with users manually altering their search approaches based on what other users claim to have worked well on their problems. DeepTuner automates this process, resulting in transfer learning search approaches with more robust search performance on new problems. This is achieved by having DeepTuner provide recommendations for improving these search approaches based on historical data in the cross-session database for representations that have been seen before.

A key challenge in recommendation is finding the right representations to recommend from. Ideally, these representation would have the same exact parameters in the representation which are evaluated in the same way. In practice, however, even just the parameters in the representation themselves can vary wildly across problems. At best, problems within the same domain, such as Petabricks, may use the same set of parameter types. However, while one representation might contain 3 *BooleanParameters*, another might contain 300. And across domains, the parameters used often won't even have the same type!

Thus finding a perfect representation is unlikely. Instead, we must settle for finding the most "similar" representation. However, defining similarity itself is a difficult problem with no single correct answer. DeepTuner uses two different metrics for measuring similarity. The first is based on parameters in the representation themselves, while the second is based on technique performance for the representation.

After determining a similar representation, the next challenge is choosing an improvement to the search approach to recommend. We note that the specification of a search approach in OpenTuner is hierarchical in nature as seen in Figure 6.1. Because of this, we need not recommend an entire search approach at once, but could instead propose modifications to subcomponents of a search approach.

[1] In machine learning, this knowledge transfer process is often referred to as transfer learning.

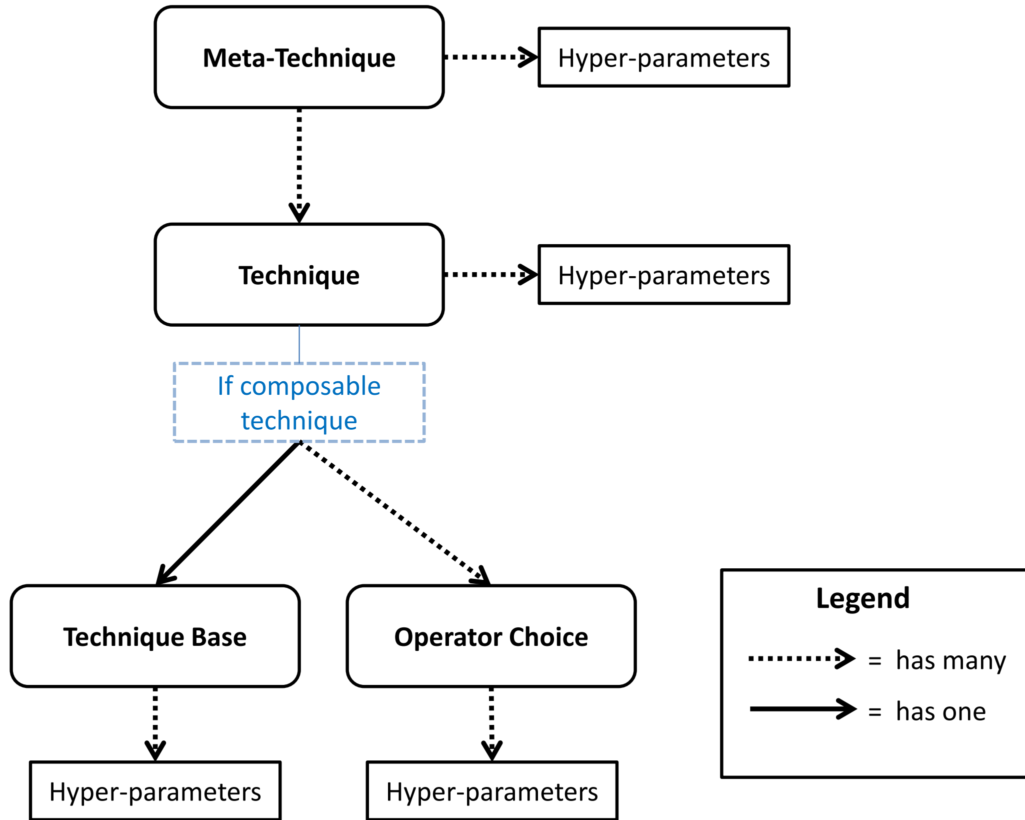


Figure 6.1: Specifying a search approach in OpenTuner
A search approach in OpenTuner consists of a meta-technique bundled with several techniques. Each of these may have several hyper-parameters to define. If the technique is a composable technique, the technique base and which operator to use for each parameter must also be specified. These may also have hyper-parameters.

The different levels of recommendation possible roughly correspond to levels of the search approach hierarchy, where Level 1 is the highest and most comprehensive level of recommendation:

Level 1. Meta-technique

At the highest level is recommendation of a fully specified meta-technique, complete with the fully specified technique bundle to be used and all hyper-parameters for the meta-technique and techniques.

Recommendation at this level is equivalent to search approach recommendation.

Level 2. Technique

Next is recommendation of a fully specified technique to use. This includes choosing not only the technique base, but also which operators to use for each parameter type. In addition, any hyper-parameter values for the technique, technique base or operators are all specified.

While this alters the technique bundle within the meta-technique, the behaviour of the meta-technique

itself is not changed.

Level 3a. Technique base

Here, given a set of operators that will be used, we recommend a technique base along with hyper-parameter values for that technique base. Essentially, given how parent configurations will be combined, we recommend an algorithm for selecting the parent configurations.

Level 3b. Operator

Here, given a particular parameter type, such as a *PermutationParamater*, we recommend an operator complete with hyper-parameter values to use. Note that operator recommendation occurs for a single parameter type.

Level 4. Hyper-parameter

At the lowest level is recommendation of only the hyper-parameter values. This can be performed for any component of the search approach, such as a particular operator or the meta-technique

Recommendation at higher levels in the hierarchy tends to have a larger potential for impacting autotuning performance as more aspects of the search approach are affected. However, this increase in performance impact potential comes with an increase in the amount of historical data needed to effectively perform the recommendation. This comes from the fact that the number of recommendation alternatives increases. For instance, for a given representation, it takes far less data to cover the space of *PermutationParameter* operators than it does to cover the space of possible search approaches.

In DeepTuner, we choose to focus on recommendation at the technique level - i.e. recommending fully specified techniques to the search approach - since we hypothesized that this would yield large performance impact without requiring an unreasonable amount of historical data generation.

6.1 The Recommendation Bandit Meta-Technique

In order to demonstrate technique recommendation using DeepTuner, we present a new class of meta-technique augmenting the original AUC bandit meta-technique in OpenTuner - the *AUC bandit meta-technique utilizing technique recommendation* or recommendation bandit meta-technique.

Given a representation, the general behaviour of the recommendation bandit meta-technique in DeepTuner during a tuning run is as follows. Note that actions in *italics* are performed by DeepTuner rather than the recommendation bandit meta-technique:

Step 0. Determine technique set

If the user specifies an initial technique set for the recommendation bandit meta-technique, skip to Step 3. Otherwise, start at Step 1 to obtain an initial technique set. We refer to a version starting at Step 1 as a database initialized or DBI recommendation bandit meta-technique.

Step 1. Submission of parameters in the representation

Send information to DeepTuner about the current representation and parameters in the representation.

Check the cross-session database to see if there is historical tuning run data for representations with similar parameters in the representation.

The similarity of parameters in the representation is quantified using a distance function. For the distance from parameter set A to parameter set B, we use the asymmetric function

$$\sum_{i \in P_A \bigcap P_B} (n_{i,A} - n_{i,B})^2 + \sum_{i \in P_A \bigcap \overline{P_B}} n_{i,A}^2 + \sum_{i \in \overline{P_A} \bigcap P_B} 1$$

where P_X is the set of parameter types in the set X, $\overline{P_X}$ is the set of parameter types not in X, and

$$n_{i,X} = \log_{10}(\text{number of parameter of type } i \text{ in } X)$$

is the order of magnitude count of the number of parameters of type i in X.

The first term accounts for distance between the number of parameters for parameter types common to both parameter sets. The second term applies a penalty for parameter types found in A but not B that grows with the number of parameters in the representation, while the final term applies a constant penalty for parameter types in B but not in A.

We choose to asymmetrically penalize parameter types in B but not in A less to reflect the facts that these parameters only make B more of a superset of A, and that a superset of parameters is more similar than a subset. This is because a superset is able to express the same range of parameter spaces as the original only with an extra dimension, whereas a subset can only express a fraction of the possible parameter spaces.

Step 2. Initial technique set recommendation

Select the most similar representation found and recommend techniques that worked well for it.

Take the top recommendations as the initial technique set. The number of techniques included in the initial set is controlled by a hyper-parameter, n whose default value is 5.

Step 3. Initialize techniques set

Instantiate the techniques in the initial technique set and initialize any populations needed by the techniques.

We note that instantiation of a recommended technique is possible given just the name as all necessary data about a technique is encoded in its name, as described in Section 5.2.

Step 4. Search space exploration

Explore the search space by using the techniques in the technique set to generate desired results to test. Like in the AUC bandit meta-technique, the selection of the next technique is performed using optimal solution to the multi-armed bandit problem outlined in Section 3.2.1.

After a set number of desired results have been submitted, pause exploration and start the recommendation phase in Step 5. The interval between recommendations is controlled by the *interval* hyper-parameter, whose default value is 50.

Step 5. Submission of technique performance

Pass the performance of the techniques that have been used on the current representation to DeepTuner.

Check the cross-session database for historical data about representations with similar performance.

We quantify technique performance for a technique as the *search affinity* using the metric

$$\frac{\text{number of new bests yielded}}{\text{number of desired results requested}}.$$

Similarity of the relative search affinity of techniques is measured by taking the the Pearson product-moment correlation coefficient, or PPMCC, on the search affinities for techniques that are mutually known between two representations. The PPMCC measures the linear correlation between two variables X and Y as a number between 1 and -1 inclusive, where 1 is total positive correlation, 0 is no correlation, and -1 is total negative correlation. For our purposes, each event is the act of using a particular technique, and the variables X and Y are the search affinity values in the two representations.

The PPMCC between X and Y is calculated by

$$\rho_{X,Y} = \frac{\text{cov}(X,Y)}{\delta_X \delta_Y} = \frac{E[(X-\mu_X)(Y-\mu_Y)]}{\delta_X \delta_Y}$$

where ρ is the PPMCC, cov is the covariance, δ_X is the standard deviation of X, μ_X is the mean of X, and E is the expected value.

Step 6. Technique recommendation

Select the most similar representation found and recommend techniques that worked well for it.

Select one of the top recommendations as a candidate technique

Step 7. Update technique set

If the candidate technique is not already initialized, initialize the candidate technique. Replace the technique in the current technique set with the worst search affinity. Go back to Step 4.

These steps are also shown in Figure 6.2.

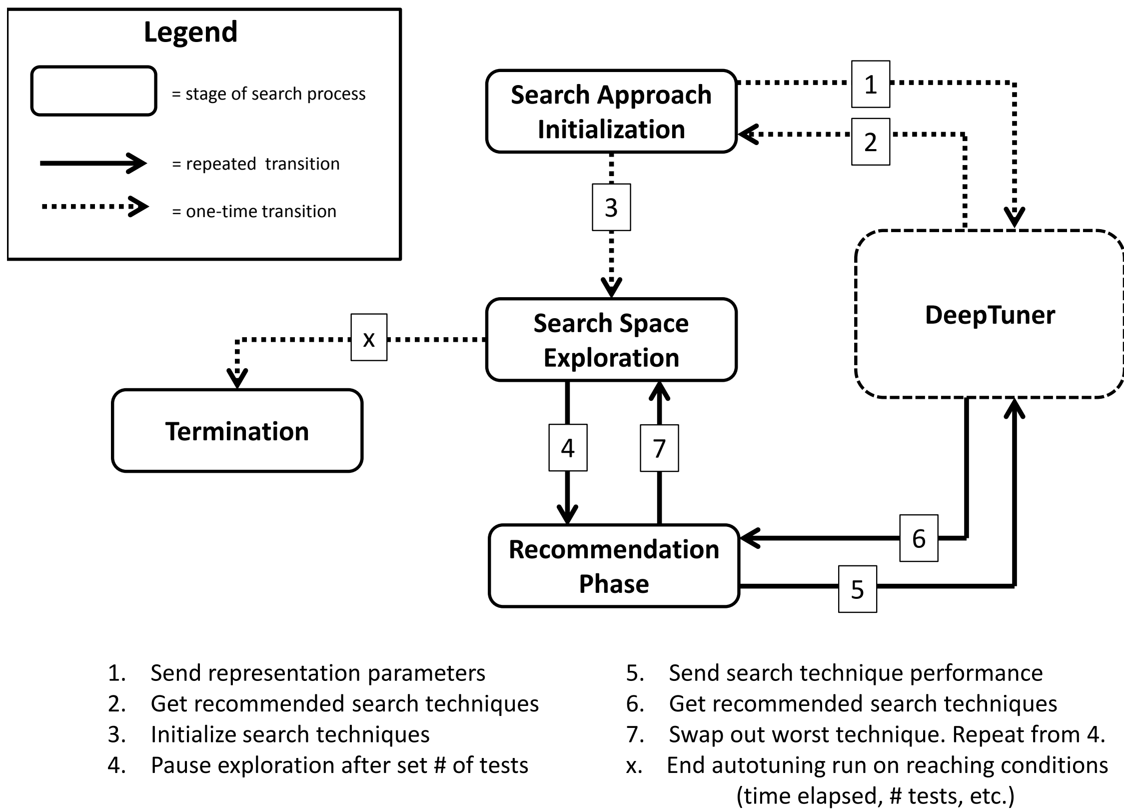


Figure 6.2: Search process for the the recommendation bandit meta-technique
Steps 1-3 deal with the initialization of the initial technique set. Step 4 occurs during search space exploration. Steps 5-7 involve the recommendation phase. The recommendation bandit meta-technique only performs steps 3-7, while the DBI recommendation bandit meta-technique performs all steps.

	search affinity		
Search Technique	dg42	br17	PBsort
PSO-OX3	0.35	0.10	0.02
GA-OX3	0.25	0.15	0.10
GA-inv	0.05	0.40	0.11
DE	0.10	0.16	0.50
PSO-inv	0.10	0.35	0.01

Table 6.1: Example performance data in the cross-session database

In this simulation, the cross-session database contains data for 3 representations across 4 techniques.

6.2 An Example of the Recommendation Process

To illustrate the recommendation process, we provide a mock simulation of a DBI recommendation bandit meta-technique that selects the top 3 initial techniques executing for a "new" representation, the TSP benchmark att48. Table 6.1 shows the data in the cross-session database for this example, which contains search affinities for 5 techniques across 3 representations - TSP benchmark dantzig42 (dg42), ATSP benchmark br17 (br17), and Petabricks benchmark sort (PBsort). Note that both TSP and ATSP benchmark representations consist of a single permutation parameter. The hyper-parameter controlling the recommendation interval is set to every 50 configurations

The DBI recommendation bandit meta-technique, referred to as "we", would proceed as follows:

Step 0. The user does not define an initial technique set when using the DBI recommendation bandit meta-technique. Thus we continue to Step 1.

Step 1. We submit the parameters in the representation of att48 (a single permutation parameter) to DeepTuner.

DeepTuner finds that it has data for dg42 and br17 in the cross-session database, both of which also have a single permutation parameter as the parameters in the representation

Step 2. DeepTuner determines that dg42 and br17 are the closest representations with a distance of 0. It recommends the top techniques from br17 after breaking the tie arbitrarily.

We select top 3 techniques: GA-inv, PSO-inv and DE.

Step 3. We instantiate these 3 techniques and initialize the populations for all 3 techniques.

representation	search affinity GA-inv	DE	PSO-inv	PSO-OX3	GA-OX3	PPMC
att48 (current)	0.09	0.10	0.30	?	?	1
dg42	0.05	0.10	0.10	0.35	0.25	0.54
br17	0.40	0.16	0.35	0.10	0.15	0.28
sort	0.11	0.50	0.01	0.02	0.10	-0.63

Table 6.2: Example search affinity similarity between representations in the cross-session database and att48
Technique search affinity similarity between representations in the example cross-session database and "known" results from the att48 representation is given by the PPMC. The PPMC is calculated using only the techniques with known results in att48 .

Step 4-1. We begin generating configurations while tracking performance of techniques. After 50 tests, we enter the recommendation phase.

Step 5-1. We submit the past performance of the techniques to DeepTuner: GA-inv tested 10 times with 1 best, PSO-inv tested 30 times with 9 bests, and DE tested 10 times with 1 best. This corresponds to search affinities of 0.1, 0.3, and 0.1 respectively.

DeepTuner examines the cross-session database for representations with similar search affinities.

Step 6-1. As seen in Table 6.2, the search affinities we know about in att48 don't match up with what's expected in br17, which we thought att48 was similar to initially in Step 2. Instead, according to the values of the PPMC, dg42 has the most similar search affinity with att48. Thus, DeepTuner recommends the best techniques from dg42. This happens to be PSO-OX3.

Step 7-1. Because PSO-OX3 is not in the current technique set, we instantiate it and initializing the population it uses. We remove GA-inv from the current technique set since it has the worst search affinity and replace it with PSO-OX3

Step 4-2. We return to generating configurations with the new technique set. This continues for 50 tests before we enter the recommendation phase again.

The DBI recommendation bandit meta-technique will then continue to execute steps 4-7 in this fashion until the termination of the tuning run.

6.3 Advantages of the Recommendation Bandit Meta-Technique

By utilizing recommendation from DeepTuner, the recommendation bandit meta-technique incorporates past tuning run knowledge to dynamically improve the initial bundle of techniques. This leads to a more robust search approach compared to those using the old AUC bandit meta-technique.

The DBI recommendation bandit meta-technique improves upon the recommendation bandit meta-technique even further. Originally, selecting an initial set of techniques was both troublesome, and non-trivial for the user. However, this step could not be neglected due to the strong impact of the initial techniques on autotuning performance for the old AUC bandit meta-technique. And, although this is mitigated in the recommendation bandit meta-technique, a poor choice of initial techniques would still adversely affect initial autotuning performance. Thus, by essentially bootstrapping the initial technique set through recommendation, the DBI recommendation bandit meta-technique removes the need for selection while providing further potential gains in autotuning performance as shown in the next chapter.

We note that the benefits of recommendation we mention are predicated on the belief that "similar" representations should interact with techniques in a similar fashion. Thus, we reiterate that the most problematic challenge in recommendation is both having "similar" representations to recommend from and having a good metric for defining what "similar" means.

Chapter 7

Results

In order to demonstrate the advantages of recommendation, we compare various default in-situ learning approaches in OpenTuner using a standard AUC bandit meta-technique to corresponding recommendation bandit meta-technique approaches with the same initial *technique set*.

7.1 Testing Protocol

To demonstrate the increased robustness of transfer learning search approaches using recommendation on new representations, we measure the autotuning performance of various search approaches on a several of representations over 3 domains - Halide, Petabricks, and TSP. Testing was performed on Lanka, a 24-node Intel Xeon E5-2695 v2 @ 2.40GHz Infiniband cluster with 12 cores and 128GB of memory per machine. When performing autotuning on a particular representation, we artificially remove historical data about the representation from the cross-session database in order to simulate the representation being new.

Each search approach is specified by a meta-technique and a set of initial techniques.

7.1.1 Meta-techniques Tested

We compare the autotuning performance for 3 different types of meta-techniques:

1. **AUC bandit meta-technique (bndt)**

   The AUC bandit meta-technique is the core meta-technique for

   search approaches in OpenTuner (see Section 3.2.1). So, we use the AUC bandit meta-technique as a performance baseline representing the default in-situ learning search approach. We use default hyper-parameter values of 500 for the bandit window size and 0.05 for the exploration-exploitation trade-off

hyper-parameter c.

2. **Recommendation bandit meta-technique (rec-bndt)**

   This type of meta-technique is based directly on the recommendation bandit meta-technique. As described in Section 6.1, this meta-technique behaves like a bndt that is augmented to periodically update its *technique set* using technique recommendations from DeepTuner. We use the same hyper-parameter values as in the bndt for the window size and c. The interval between recommendations is set to every 25 desired results

3. **Recommendation meta-technique (rec)**

   This type of meta-technique is a variation of the rec-bndt where the hyper-parameter c has been set to always favour exploration. As a result, testing resources end up allocated evenly across the *technique set*, effectively eliminating the bandit. These act as a control to disentangle the effects of the bandit from the recommendation. The interval between recommendations is also set to every 25 desired results.

We will refer to these three meta-technique types as the bndt, rec-bndt, and rec, respectively.

Initial Technique Sets

We compare the different meta-techniques over 4 different kinds of initial *technique sets*. The first 3 kinds are static *technique sets* that are chosen to reflect typically used search approaches in OpenTuner. The last kind uses DeepTuner's technique recommendation to dynamically select an initial *technique set* for the given representation.

1. **Technique Set "A" (A)**

   The initial *technique set* of the default search approach in OpenTuner used when no search approach is specified. This uses a Nelder-Mead search hill climber, differential evolution, and two variants of greedy mutation.

2. **Technique Set "pso-ga-de" (pgd)**

   The *technique set* of a search approach in OpenTuner that is often anecdotally recommended for use. As the name suggests, this set consists of three variants of particle swarm optimization, three genetic algorithms, and differential evolution.

3. **Technique Set "D" (D)**

The *technique set* for a custom search approach in OpenTuner. This consists of a Nelder-Mead search hill climber, differential evolution, a pattern search hill-climber, and one instance of particle swarm optimization.

4. **Database Initialized (dbi)**

   The initial *technique set* is chosen through recommendation from DeepTuner based on the parameters in the representation being tuned. This uses the DBI recommendation bandit meta-technique mentioned in Section 6.1. The value of the hyper-parameter n controlling how many techniques to choose is set to 5.

Search Approaches Tested

Combining the 3 types of meta-technique with the 4 initial *technique sets* gives the 12 final search approaches. These labels are summarized in Table 7.1.

We note that bndt-A is the default in-situ learning search approach in OpenTuner.

Search Approach Label	Meta-technique	Initial Technique Set	Learning Type
bndt-A	bndt	A	in-situ learning
bndt-pgd	bndt	pgd	in-situ learning
bndt-D	bndt	D	in-situ learning
bndt-dbi	bndt	dbi	transfer learning
rec-bndt-A	rec-bndt	A	transfer learning
rec-bndt-pgd	rec-bndt	pgd	transfer learning
rec-bndt-D	rec-bndt	D	transfer learning
rec-bndt-dbi	rec-bndt	dbi	transfer learning
rec-A	rec	A	transfer learning
rec-pgd	rec	pgd	transfer learning
rec-D	rec	D	transfer learning
rec-dbi	rec	dbi	transfer learning

Table 7.1: Labels for tested search approaches

7.2 Search Approach Performance Comparisons

We compare autotuning performance for the 12 search approaches on nine different representations spread over three domains. The autotuning performance results were averaged by taking median values over at

least 75 autotuning runs. The error bars show the 20th and 80th percentiles.

7.2.1 Halide

Halide is a domain specific language and compiler specifically designed for high-performance image processing pipelines [25]. Halide separates the algorithmic intent of the image processing kernels from the *schedule* by which the algorithm is executed on the machine. Using the right schedule for the underlying architecture balances the trade-offs between parallelism and locality in the execution, resulting in faster execution of up to an order of magnitude from a naive implementation. Thus, the autotuning problem in Halide is to search the configuration space of potential schedules for a given pipeline to yield high performance schedules on the given architecture.

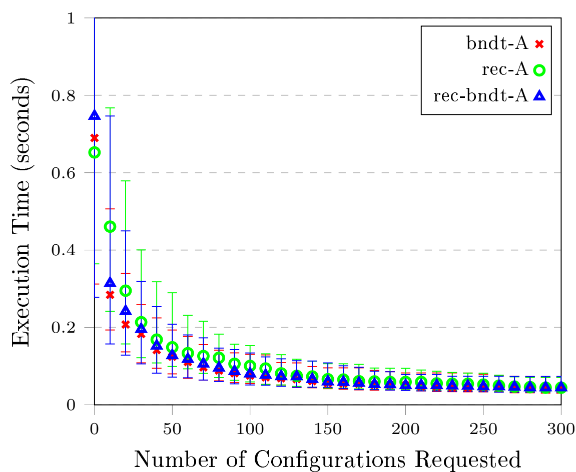
Figure 7.1 shows the performance of the various search approaches for the bilateral grid benchmark in Halide grouped by initial *technique set* and Figure 7.2 shows final execution times at the end of autotuning. For each initial *technique set*, the rec and rec-bndt perform on-par with their in-situ learning bndt counterpart. For *technique set* D in Figure 7.3c , the rec and rec-bndt seem to perform slightly better than the bndt, although this difference is small relative to the variance.

We hypothesize that the search approaches using recommendation fail to show a significant advantage over the traditional in-situ learning search approaches due to "simplicity" of the bilateral grid representation. As seen in Figure 7.3, almost all of the search approaches converge very quickly towards a near optimum and level out within a relatively small number of tested configurations. The speed of convergence results in very little time for recommendation of techniques to make a large impact. Furthermore, the lack of significant performance differences across initial *technique sets* suggest that the choice of technique has a small impact on autotuning performance. This may be because the majority of techniques are good enough.

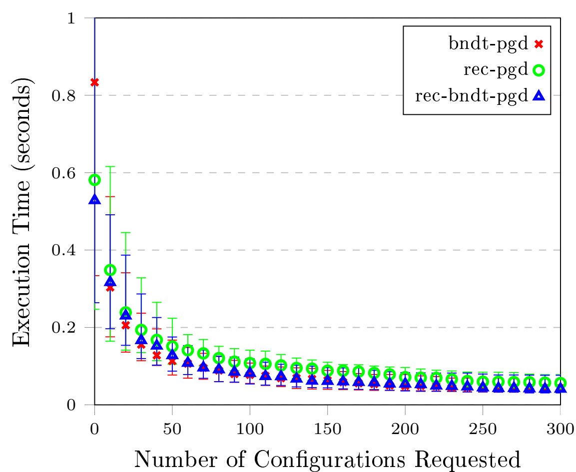
7.2.2 Petabricks

Petabricks is a language and compiler that incorporates algorithmic choices as a first class construct in the language[4]. Petabricks allows the programmer to define multiple implementations of different algorithms to solve the same problem. These can be stitched together into *poly-algorithms* using different individual algorithms on each level or part of the data. These are synthesized from *algorithmic selectors* that map input sizes to an algorithmic choice to use for the input size.

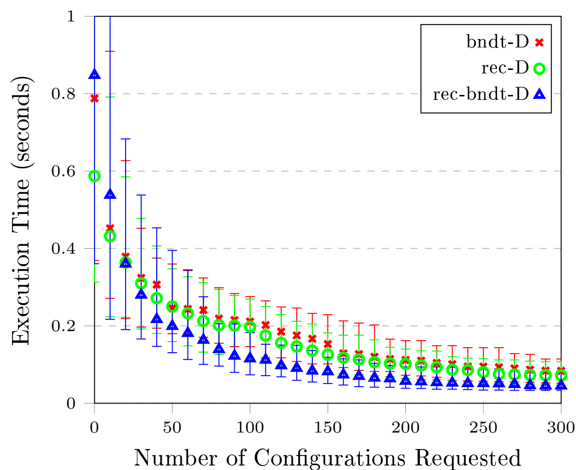
For a given Petabricks program, there may be several such choices, along with lower level optimization choices like block sizes or thread counts. The autotuning problem in Petabricks is to search the space of poly-algorithms and low level optimizations for a program in order to minimize execution time of the final



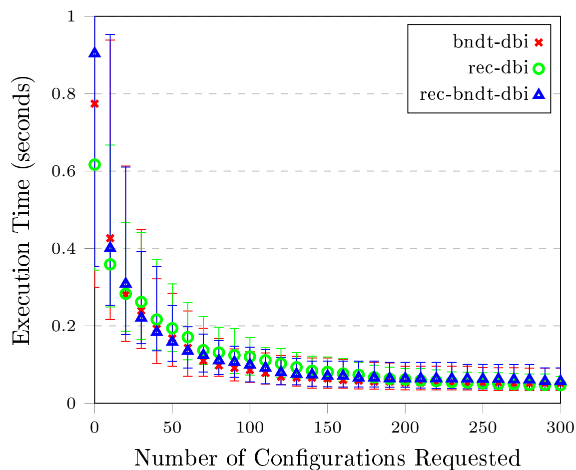
(a) technique set "A"



(b) technique set "pso-ga-de"



(c) technique set "D"



(d) database initialized technique set

Figure 7.1: Search approach performance in Halide bilateral grid by initial technique set
Best execution time achieved as a function of number of configurations tested by the search approach, aggregated over 75 tuning runs. Lower execution time is better.

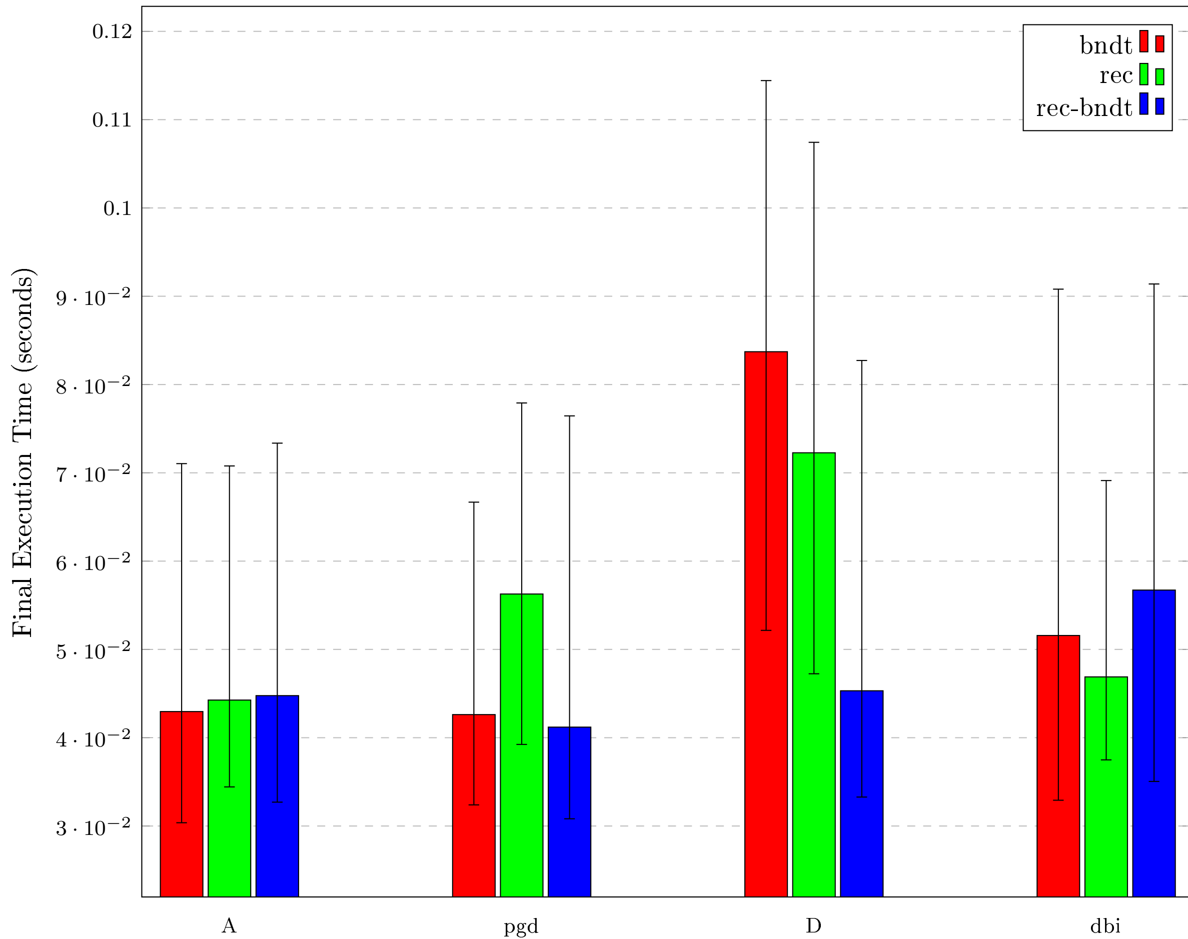


Figure 7.2: End of autotuning execution times for the Halide bilateral grid benchmark
Best execution times found by search approaches after 300 configurations tested. The search approaches are grouped by initial *technique set*. Values shown are medians over 75 tuning runs.

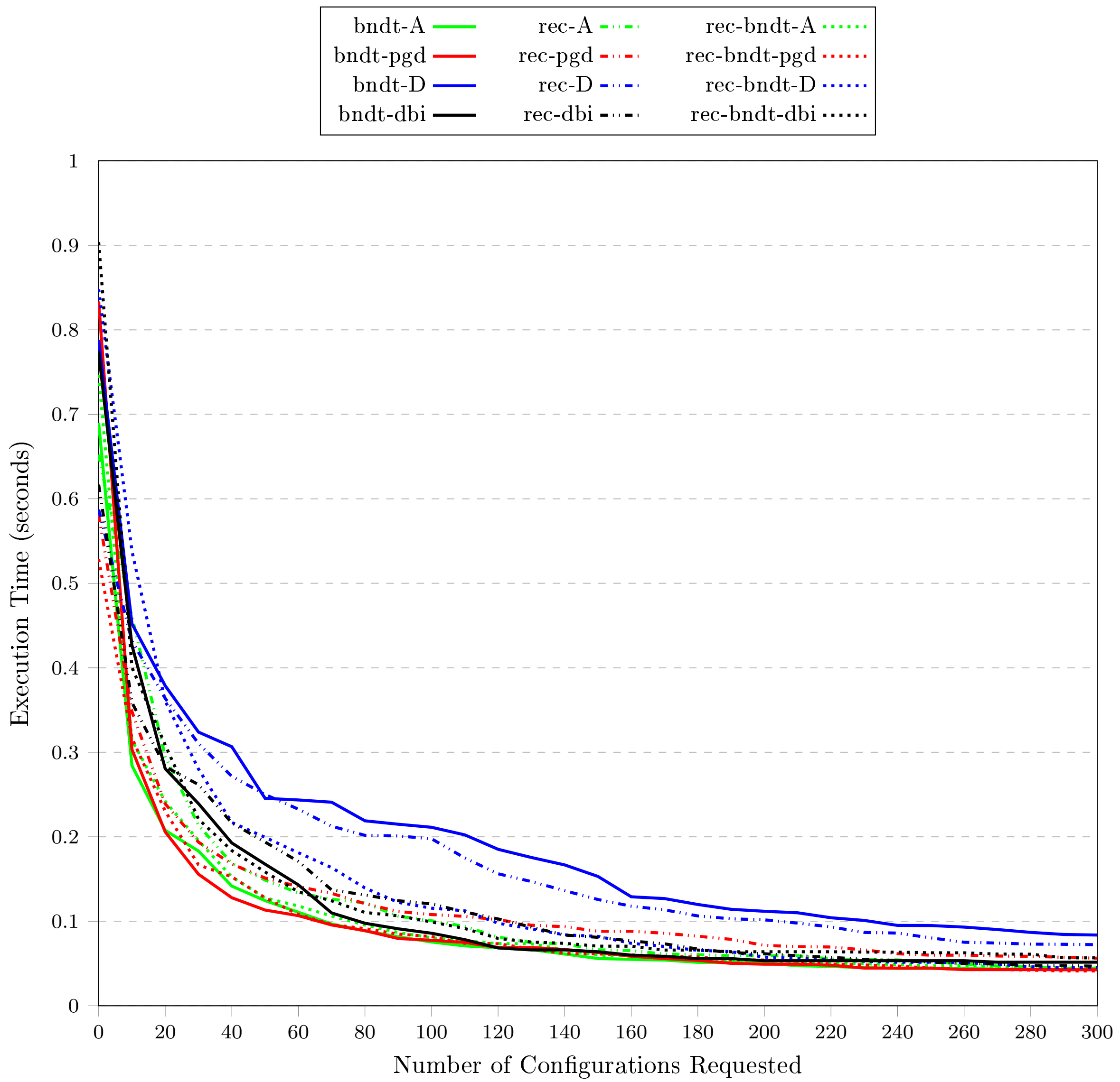
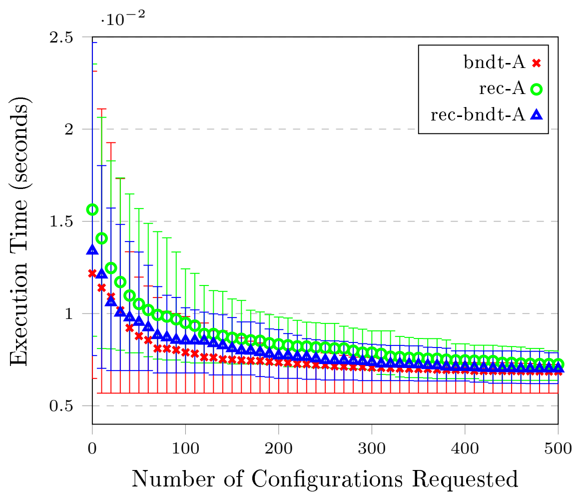
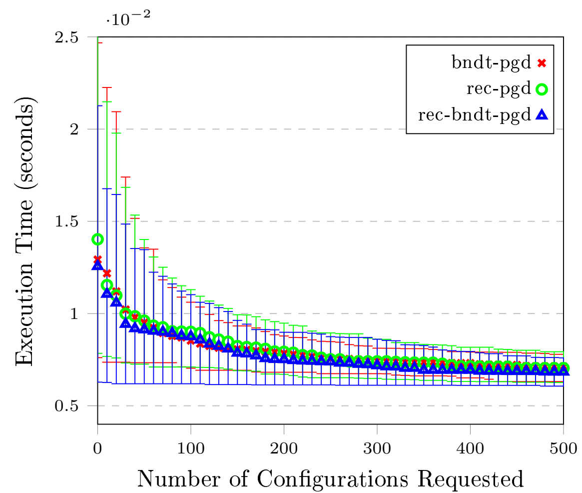


Figure 7.3: Search approach performance in the Halide bilateral grid benchmark
The lines represent the median values over 75 tuning runs.

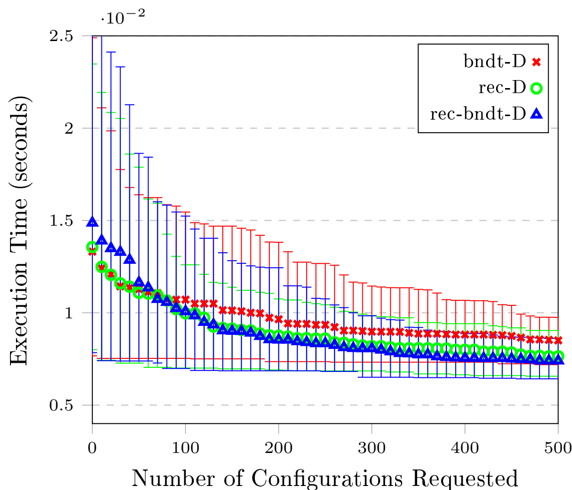
specified program.



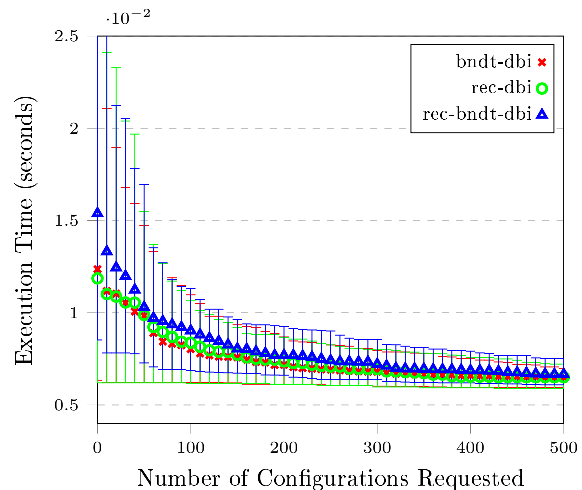
(a) technique set "A"



(b) technique set "pso-ga-de"



(c) technique set "D"



(d) database initialized technique set

Figure 7.4: Search approach performance in Petabricks matrix approximation by initial technique set
Best execution time achieved as a function of number of configurations tested by the search approach, aggregated over 150 tuning runs. Lower execution time is better.

Figures 7.4, 7.4, and 7.6 show search approach autotuning performances for the matrix approximation problem in Petabricks. As before, differences in performance between search approaches using the same initial *technique set* are often small relative to the variation seen across tuning runs. More significant performance differences are shown in Figure 7.6a between bndt-A and rec-A and in Figure 7.6c between bndt-D and both rec-bndt-D and rec-D.

The default bndt-A appears to yield better performance than rec-A. The majority of this difference is likely due to the lack of the bandit in rec-A rather than negative effects from recommendation, as the rec-

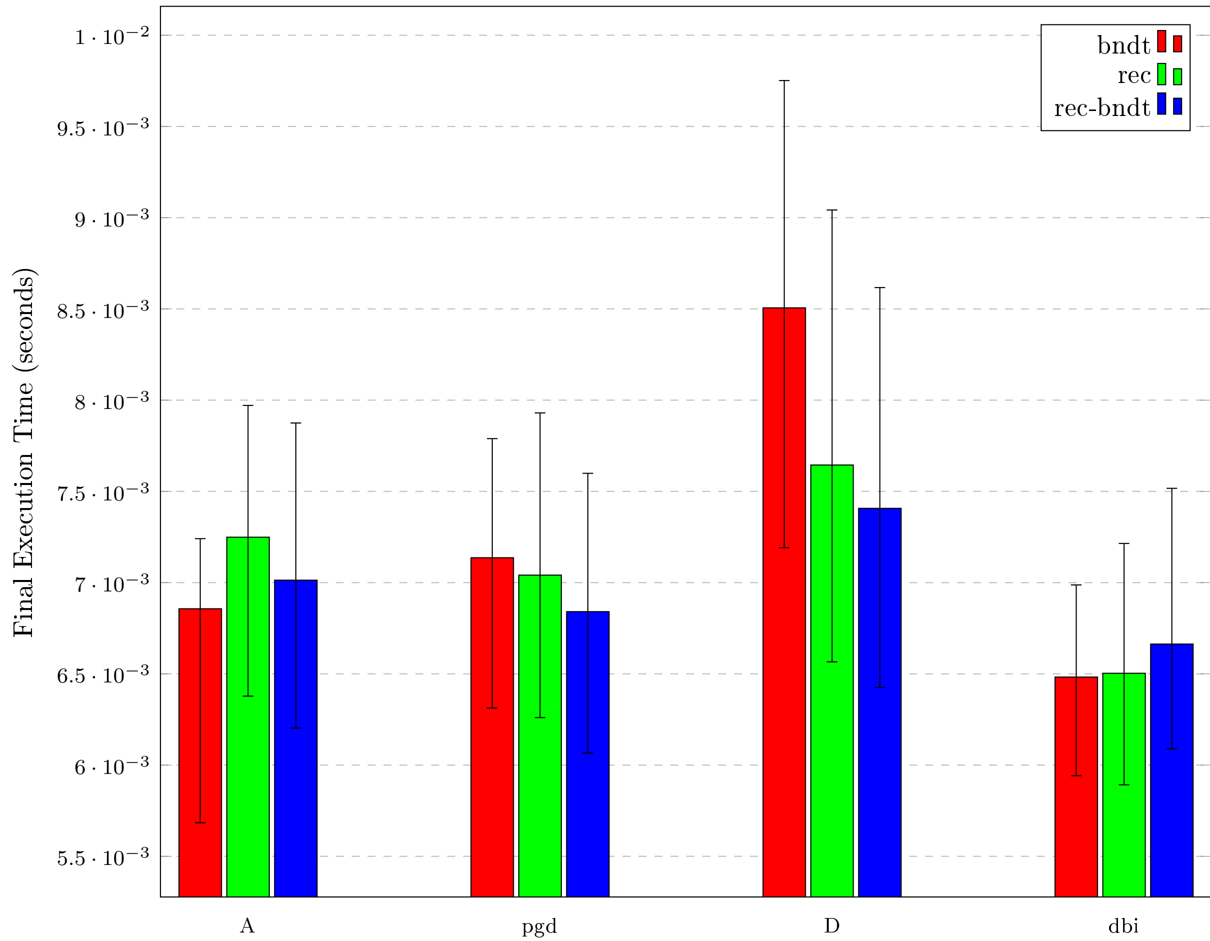


Figure 7.5: End of autotuning execution times for the Petabricks matrix approximation benchmark
Best execution times found by search approaches after 500 configurations tested. The search approaches are grouped by initial *technique set*. Values shown are medians over 150 tuning runs.

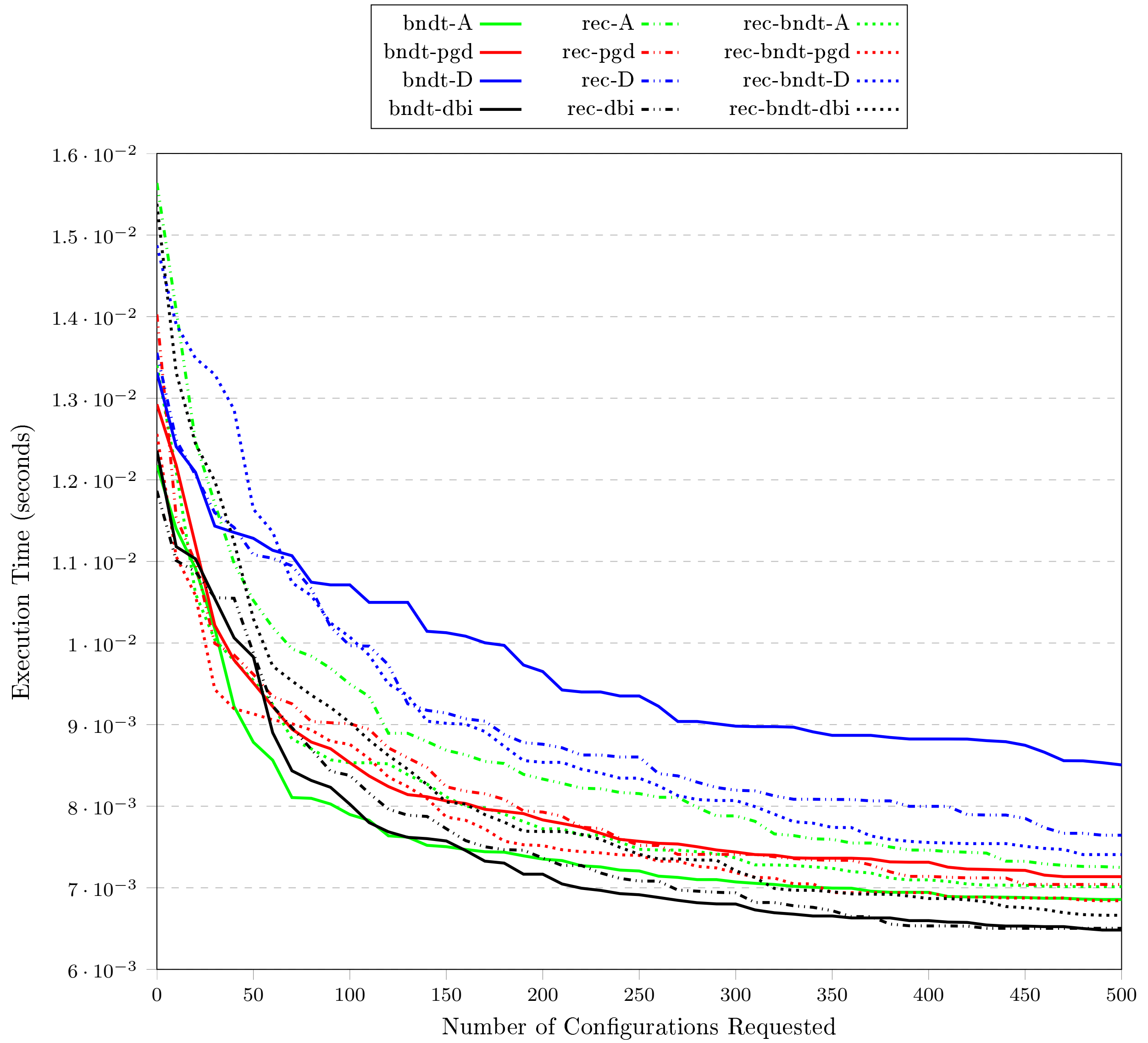


Figure 7.6: Search approach performance in the Petabricks matrix approximation benchmark
The lines represent the median values over 150 tuning runs.

bndt-A shows similar performance to bndt-A. Furthermore, the default bndt-A is one of the top performing search approaches, suggesting the initial *technique set* is already near optimal, which leaves little room for improvement by recommendation.

On the other hand, technique set D seems to be much less optimal for matrix approximation. As a result, both rec-D and rec-bndt-D show relatively significant performance gains over bndt-D within the first 100 configurations tested and at the end of autotuning.

Overall, the search approaches using the dbi technique set resulted in the best autotuning performance. These also tended to achieve the best final execution times.

In contrast, the bndt-dbi yielded the slowest convergence for Strassen, as seen in Figure 7.8. This suggests that the initial technique recommendation based on the parameters in the representation, while often fairly accurate, may not always be accurate. However, these variations between search approaches in performance may just be due to the high variation in performance across tuning runs in Strassen, as seen in Figure 7.7.

For all initial *technique sets*, we see that incorporating recommendation does not harm the performance of any of the bndt based search approaches. In fact, for bndt-D and bndt-pgd, utilizing recommendations even improves final autotuning perfomance and the convergence speed slightly.

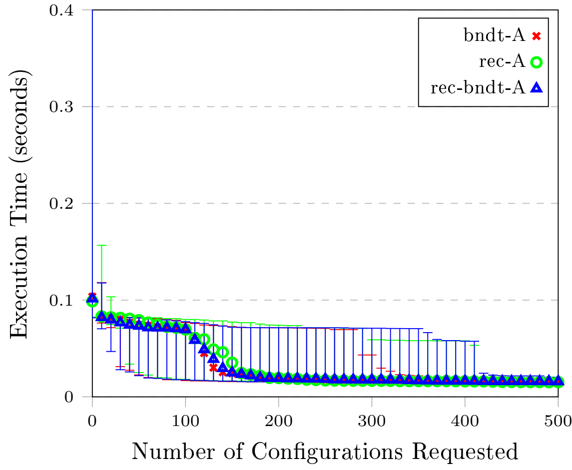
7.2.3 Travelling Salesman Problem

In the travelling salesman problem (TSP), the goal is to find the shortest route visiting all cities given the distances between the cities. This translates nicely to an autotuning problem of finding the ordering of cities that gives the shortest route for a specified distance matrix, where the ordering is represented as a single permutation parameter. The benchmarks in the TSP autotuner are taken from various TSP and asymmetric TSP (ATSP) problems in the TSPLIB library [1]. General information about the benchmarks is summarized in Table 7.2. We compare search approach performance over a subset of these benchmarks.

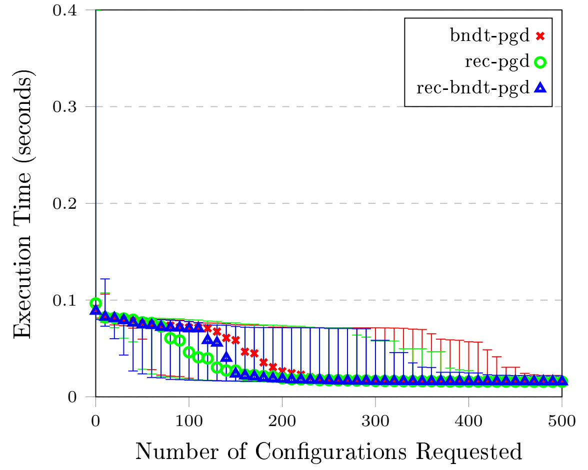
Symmetric TSP

Figures 7.9 and 7.10 show results for two symmetric TSP benchmarks att48 and dantzig42, respectively. Both benchmarks show similar results.

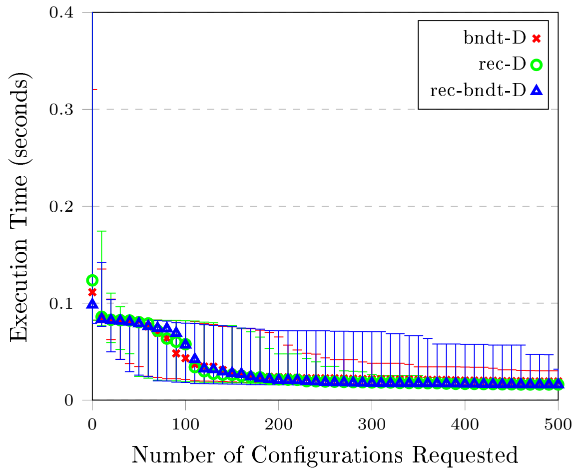
For initial technique sets A, D, and pgd, the autotuning performance between the three types of meta-techniques is initially similar. However, the bndt is consistently outperformed by both the rec-bndt and the rec later by a significant margin. The performance of the rec and rec-bndt search approaches begin to diverge from those of the bndt at about 500 tested configurations, or 20 iterations of recommendation. At this point, it appears there is sufficient technique performance data for DeepTuner to successfully match



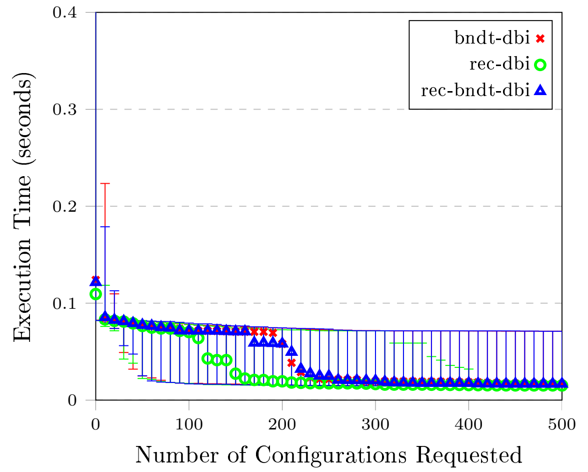
(a) technique set "A"



(b) technique set "pso-ga-de"



(c) technique set "D"



(d) database initialized technique set

Figure 7.7: Search approach performance in Petabricks Strassen by initial technique set
Best execution time achieved as a function of number of configurations tested by the search approach, aggregated over 150 tuning runs. Lower execution time is better.

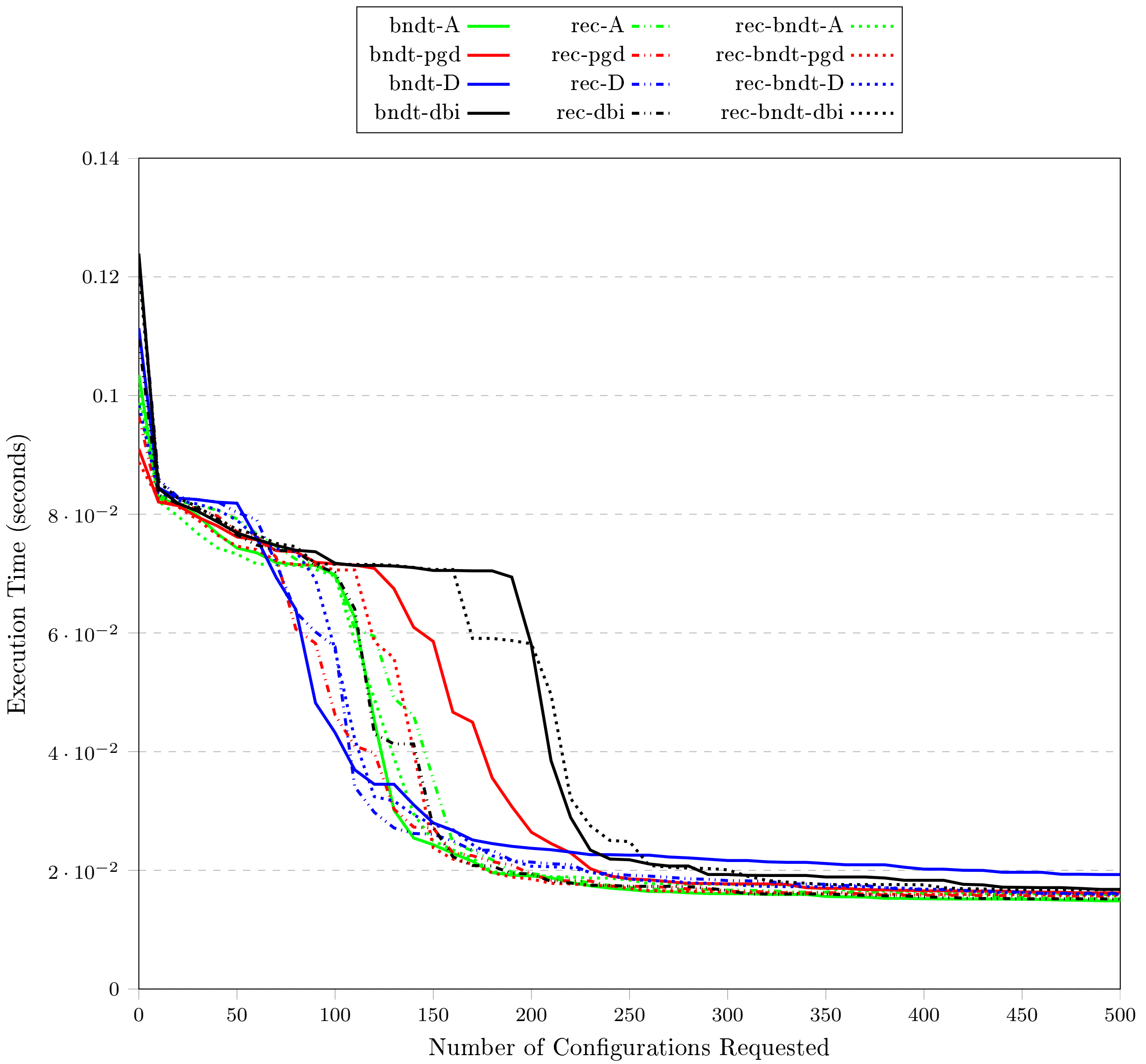
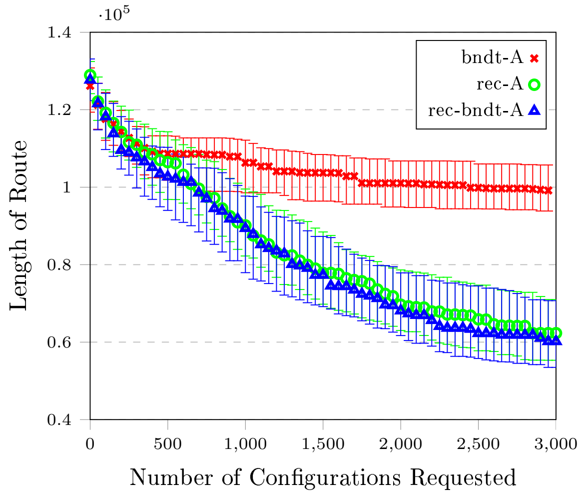
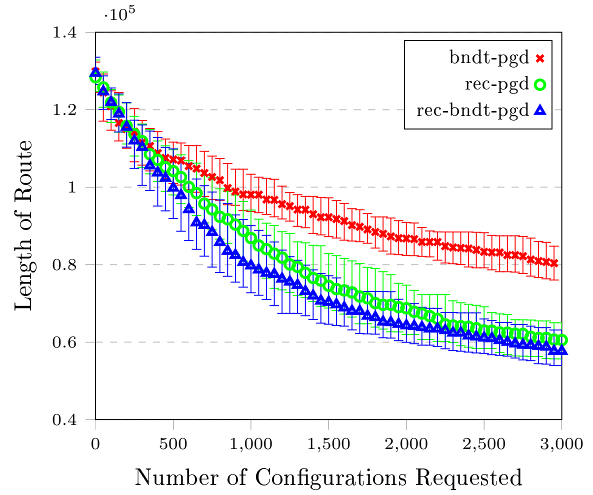


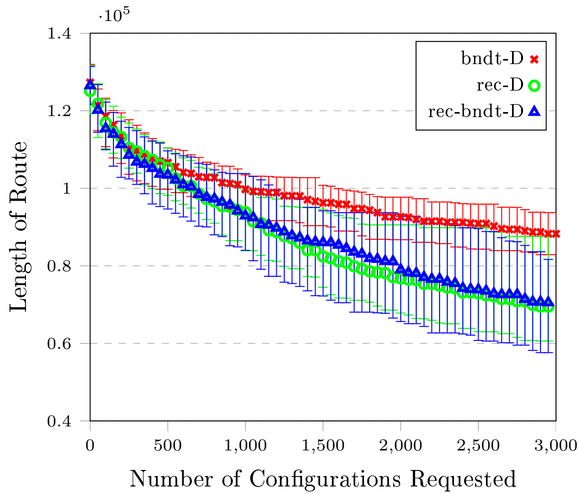
Figure 7.8: Search approach performance in the Petabricks Strassen benchmark
The lines represent the median values over 150 tuning runs.



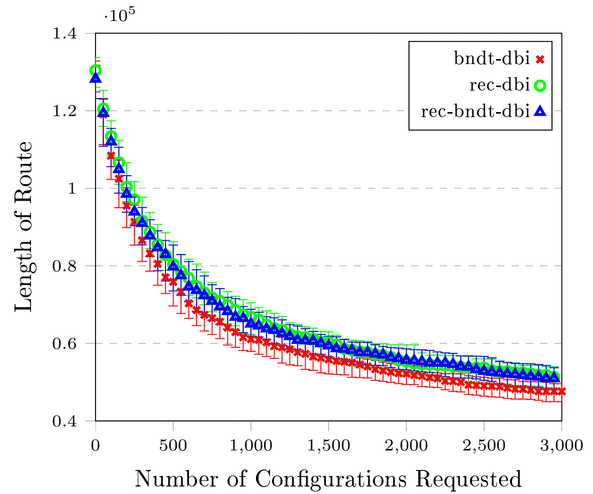
(a) technique set "A"



(b) technique set "pso-ga-de"

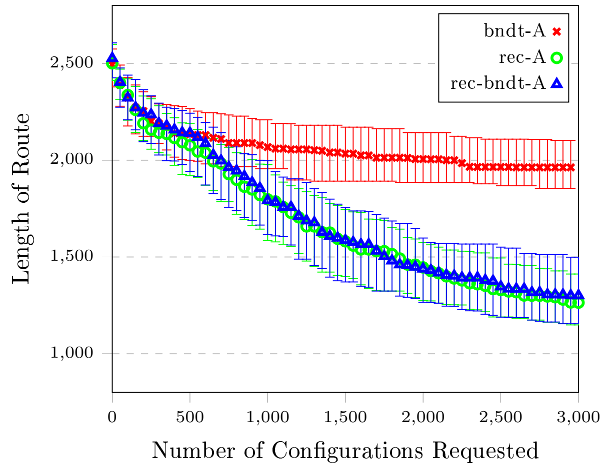


(c) technique set "D"

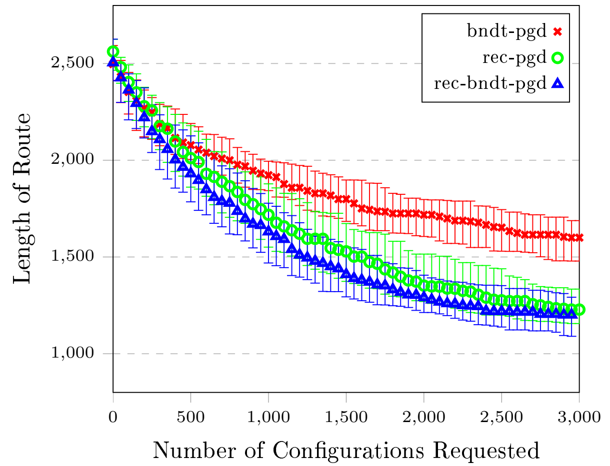


(d) database initialized technique set

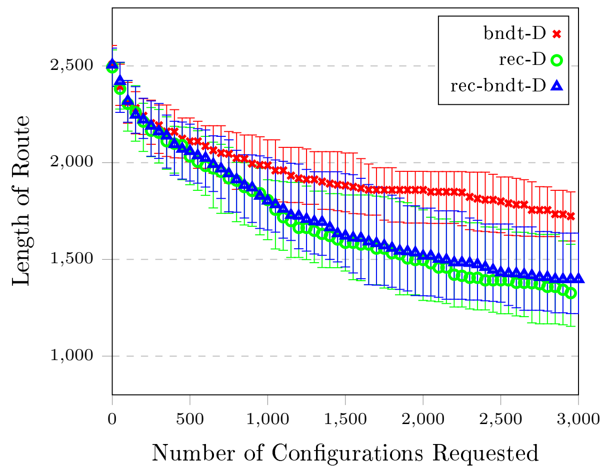
Figure 7.9: Search approach performance in TSP att48 by initial technique set
Shortest route length found as a function of number of configurations tested by the search approach, aggregated over 75 tuning runs.



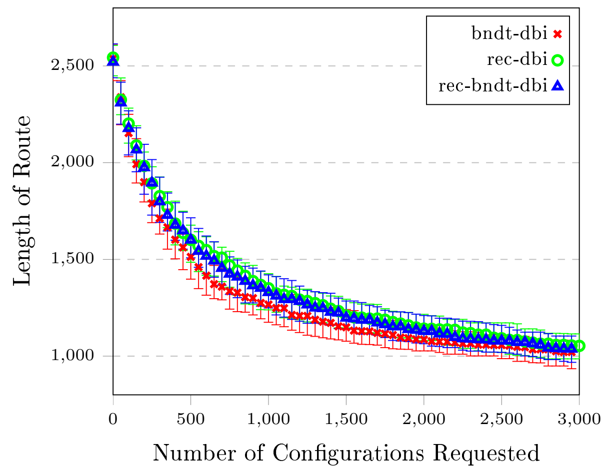
(a) technique set "A"



(b) technique set "pso-ga-de"



(c) technique set "D"



(d) database initialized technique set

Figure 7.10: Search approach performance in TSP dantzig42 by initial technique set
Shortest route length found as a function of number of configurations tested by the search approach, aggregated over 75 tuning runs.

Benchmark Name	Symmetric?	Number of Cities	Best Known Route Length
att48	yes	48	10628
brazil58	yes	158	25395
brg180	yes	180	1950
dantzig42	yes	42	699
gr120	yes	120	6942
gr48	yes	48	5046
si175	yes	175	21407
swiss75	yes	75	1273
br17	no	17	39
ftv170	no	171	2755
ftv44	no	45	1613
p43	no	43	639
rgb323	no	323	1326
ry48p	no	48	14422

Table 7.2: Benchmarks in the TSP autotuner
The benchmarks and best known solutions are taken from [1]. Route lengths for symmetric TSP problems are optimal.

the current representation to a similar representation in the cross-session database. Because there is little difference between the performance of the rec-bndt and rec in these cases, we conclude that the increase in autotuning performance is almost entirely due to the effects recommendation.

The large period of time before the effects of recommendation kick in could be caused by skewing of performance data. The bias would be introduced in the early stages of autotuning due to the use of new bests in the performance metric. Because there are very few tested configurations initially, it is much easier for techniques to discover new bests during the early stages of autotuning. As a result, techniques in the initial set end up with an inflated search affinity, resulting in faulty recommendations. As the autotuning session continues, the inflated search affinity values normalize leading to more accurate recommendations.

With the dbi *technique set*, the bndt performs slightly better than both the rec and rec-bndt. This suggests that the initially recommended set of technique based on the parameters in the representation was already highly optimal for the current representation, which is likely as the most similar representation would be a different TSP instance. As a result, further recommendation could hurt autotuning performance by swapping in a less optimal technique or by destroying the state of the technique that is swapped out.

For both att48 and dantzig42, we see in Figures 7.11, 7.12, 7.13, and 7.14 that the three in-situ learning search approaches - bndt-A, bndt-D, and bndt-pgd - yield the poorest autotuning performance. In fact,

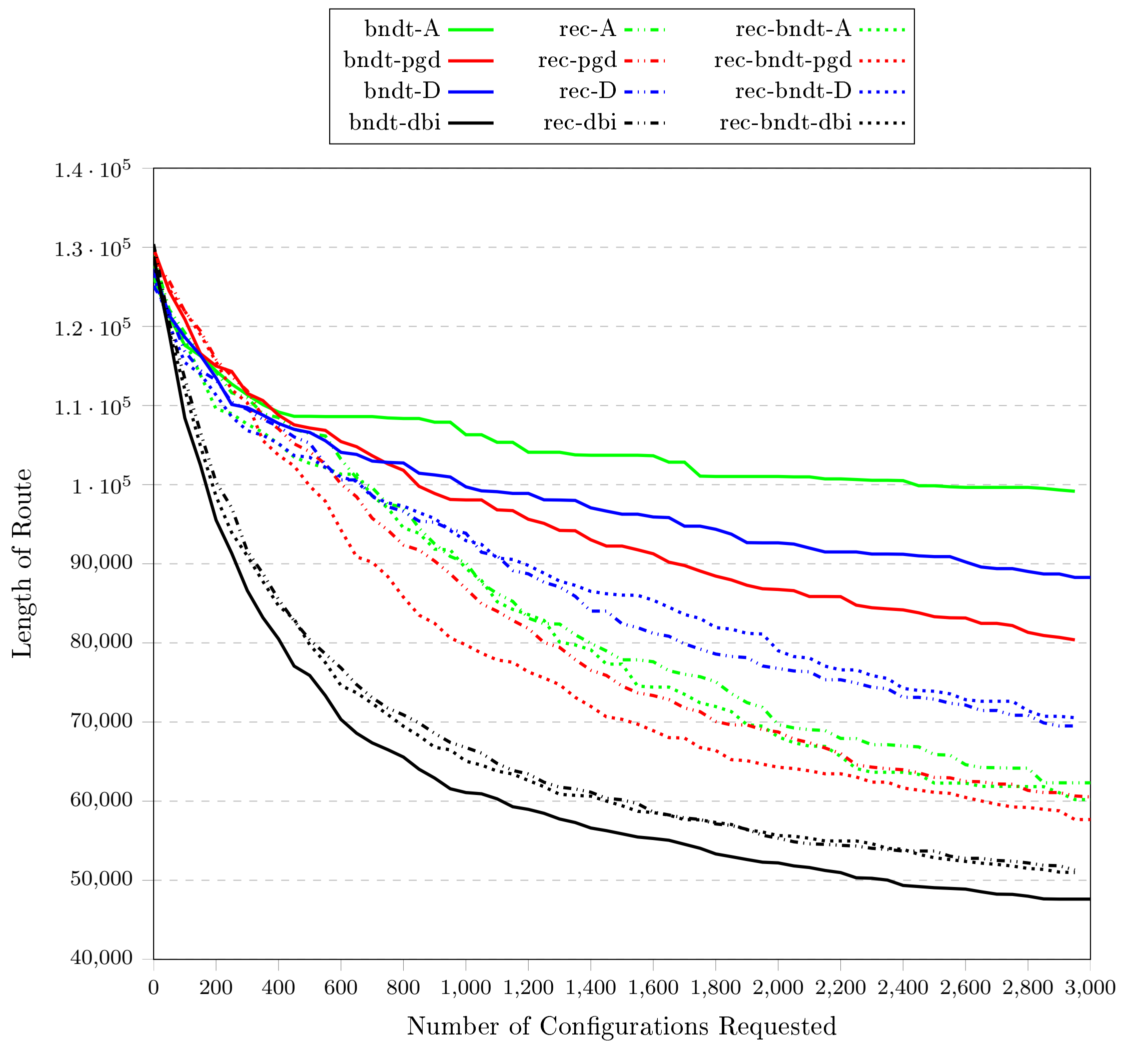


Figure 7.11: Search approach performance in TSP att48
The lines represent the median values over 75 tuning runs.

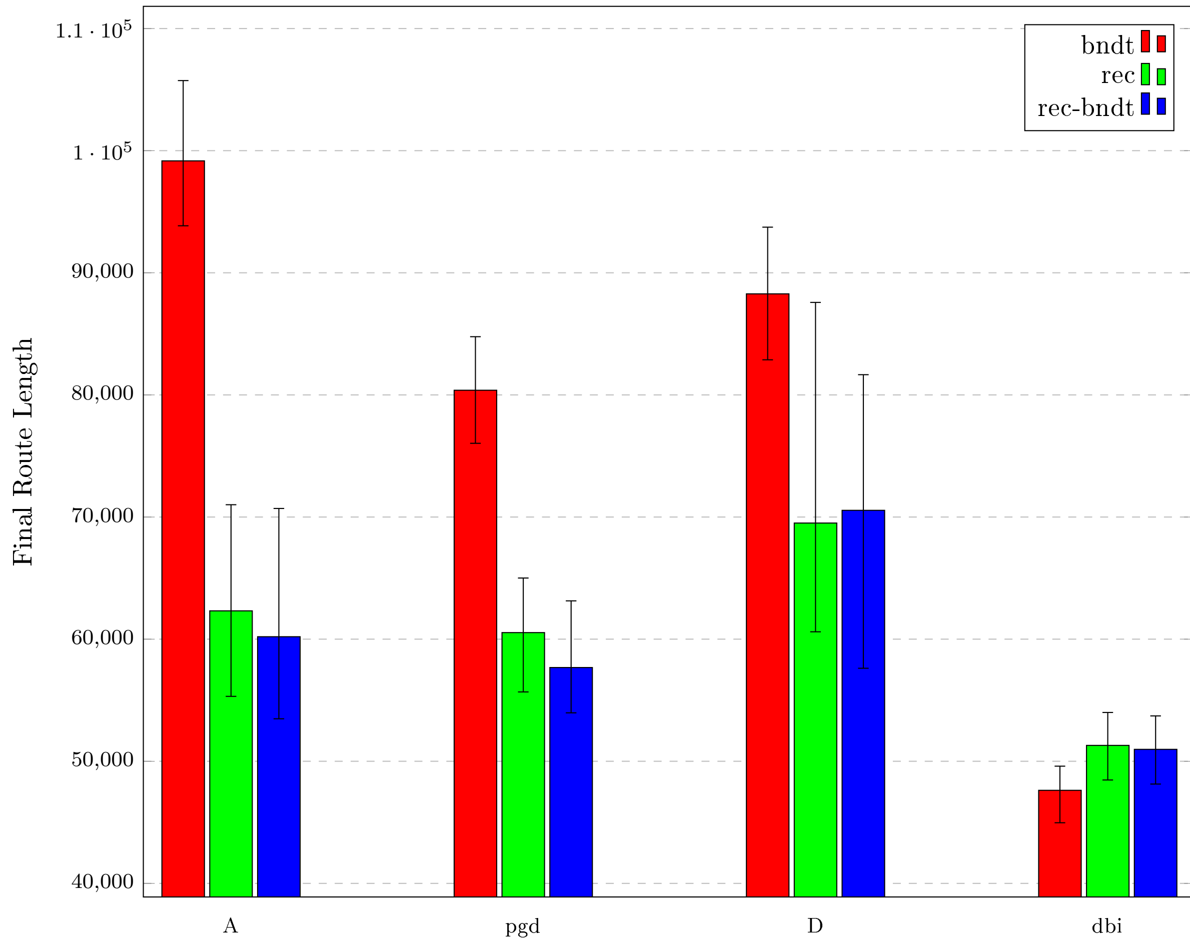


Figure 7.12: End of autotuning best route lengths for the TSP att48 benchmark
Best route lengths found by search approaches after 3000 configurations tested. The search approaches are grouped by initial *technique set*. Values shown are medians over 75 tuning runs.

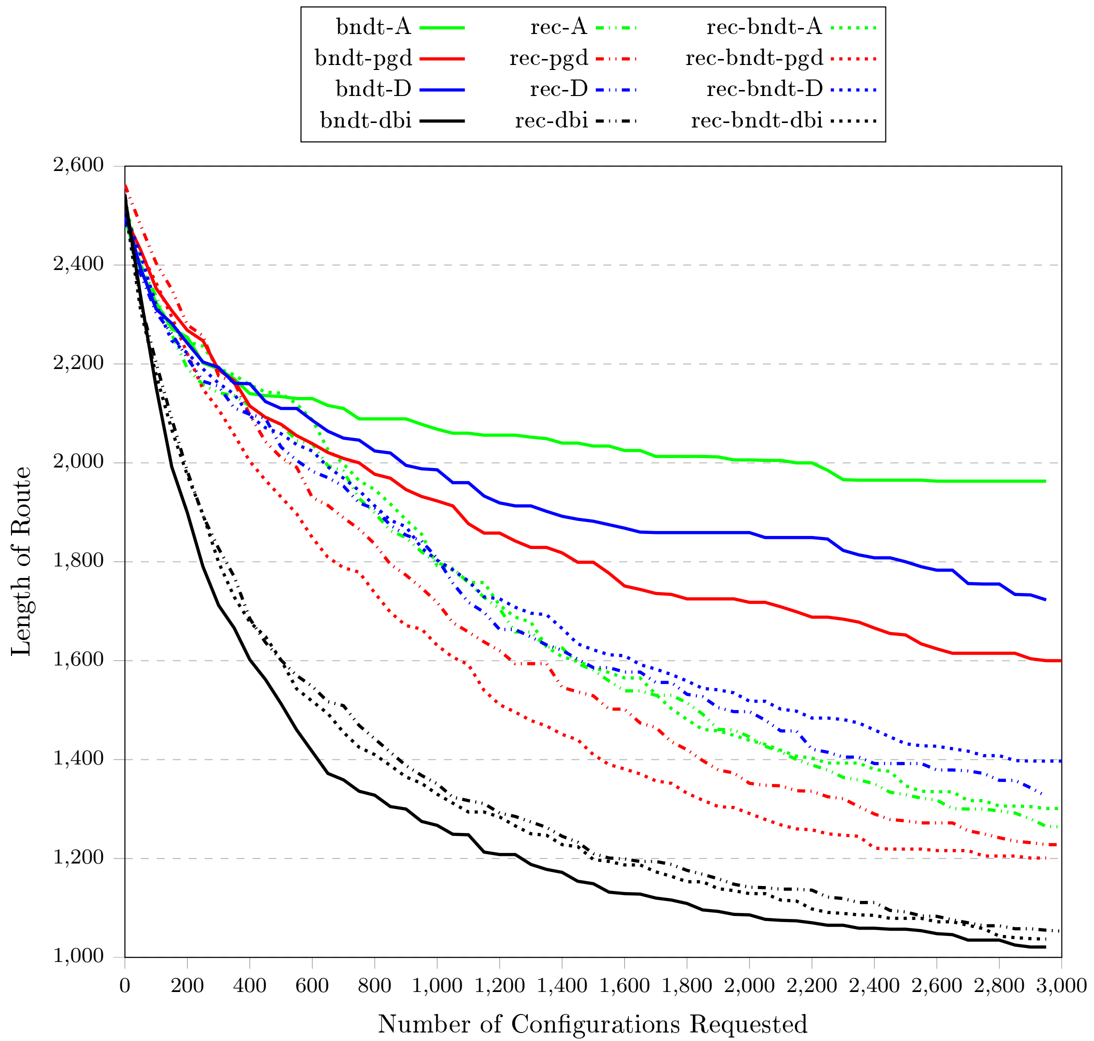


Figure 7.13: Search approach performance in TSP dantzig42
The lines represent the median values over 75 tuning runs.

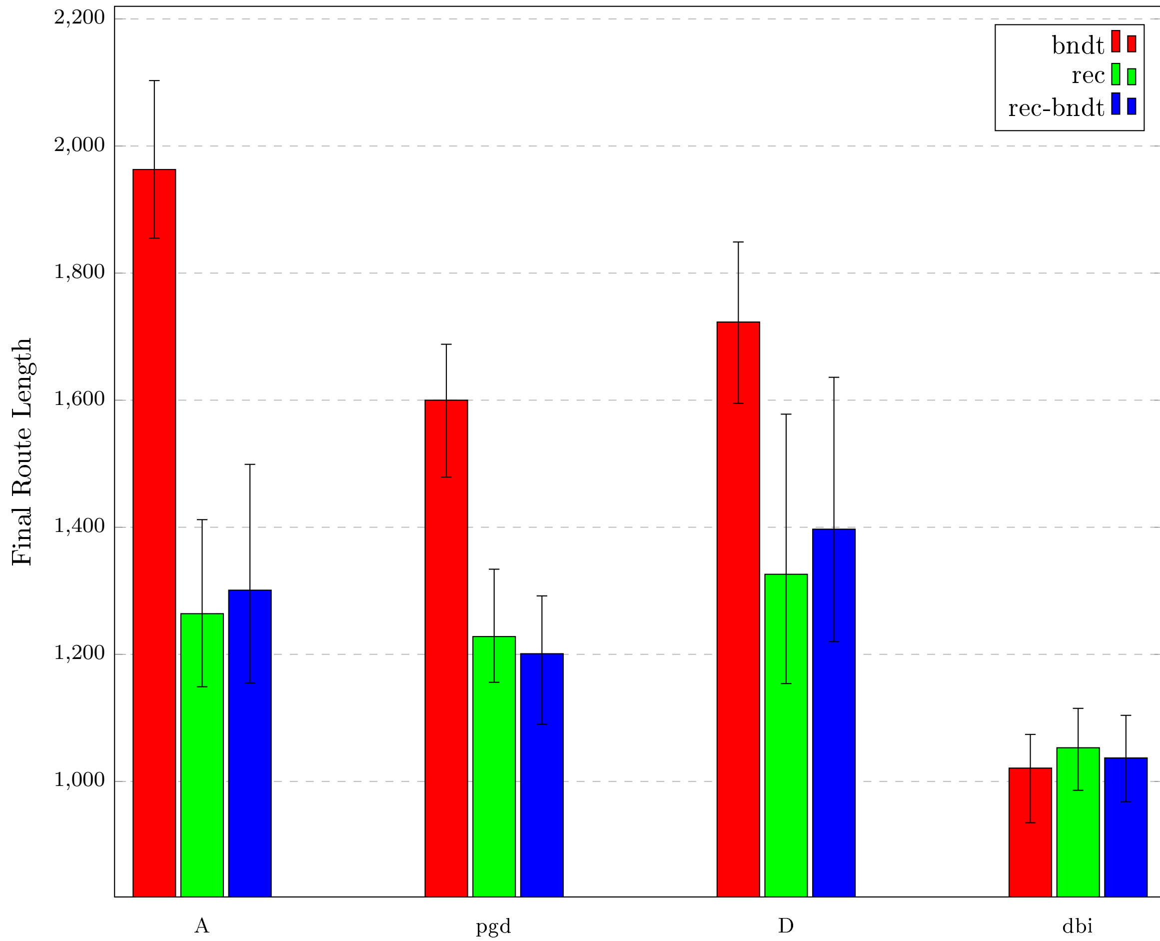


Figure 7.14: End of autotuning best route lengths for the TSP dantzig42 benchmark
Best route lengths found by search approaches after 3000 configurations tested. The search approaches are grouped by initial *technique set*. Values shown are medians over 75 tuning runs.

bndt-A, which is the default search approach in OpenTuner, is the worst of all twelve search approaches tested by a significant margin. While the anectodally suggested bndt-pgd fares slightly better than the other two in-situ learning search approaches, it is still outperformed by every single transfer learning search approach.

We see that for fixed initial *technique sets*, the rec and rec-bndt search approaches achieve much better final route lengths than their bndt counterparts. Overall though, the search approaches using a dbi *technique set* show the best performance.

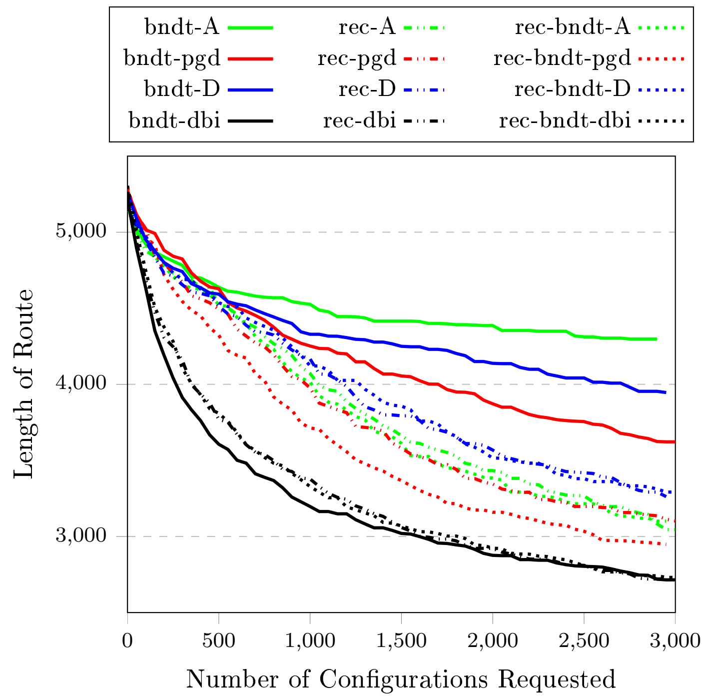
Asymmetric TSP

Figure 7.15 shows performance results for the ATSP benchmarks ftv44 and ftv170, while figure 7.16 shows the best route lengths discovered at the end of autotuning. The results are consistent with results for symmetric TSP benchmarks, with the transfer learning search approaches outperforming the three in-situ learning search approaches and bndt-A showing the worst performance. As before, while the autotuning performance of the bndt for the fixed initial *technique sets* is initially similar to the other two meta-technique types, the performance of the recommendation meta-techniques begins to diverge after around 500 configurations tested.

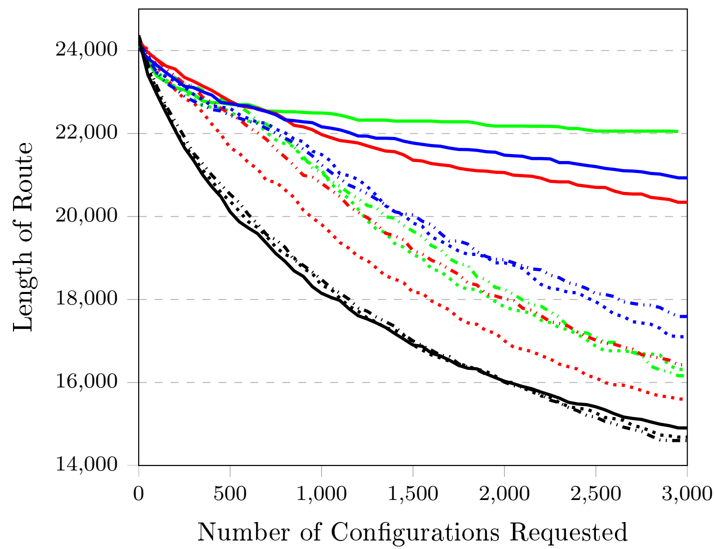
The search approaches with a dbi *technique set* again show the best overall and final performance. However, unlike with the TSP benchmarks, the bndt is eventually outperformed by both the rec-bndt and rec, despite showing better performance initially. This can seen more clearly in Figure 7.17, where the length of the tuning runs have been doubled to 6000 requested configurations.

This suggests that despite being very similar, there are still differences across TSP representations in terms of which techniques yield the best search performance. The initial technique recommendation by parameters in the representation, though relatively good, failed to yield the very best possible *technique set* as the various TSP representations all consist of a single permutation parameter and are indistinguishable in the dimension of parameters in the representation. As a result, the rec-bndt and rec are eventually able to obtain better performance than the bndt by recommending techniques for the most similar TSP representation in the technique search affinity dimension. The divergence in performance appears to occur around 2000 configurations.

We note that the number of tested configurations required to see this fine-tuning of the *technique set* is larger than the number needed to see the effects of recommendation starting with, say, technique set A. This is due to the fact that distinguishing between TSP representations, which are relatively similar in terms of technique search affinity, inherently requires more data than distinguishing between representations in different domains.

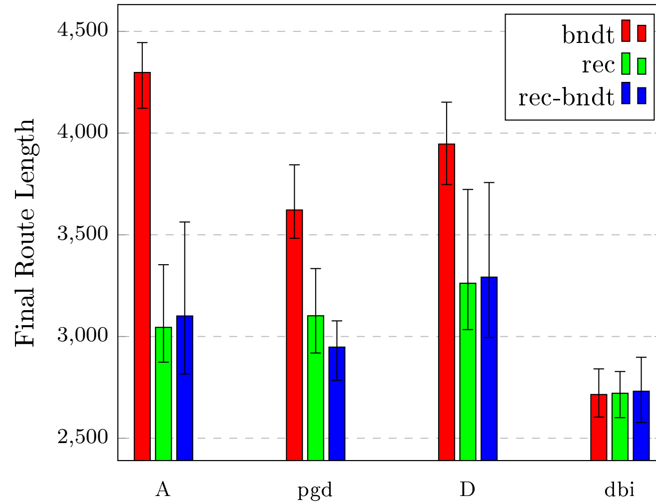


(a) ftv44

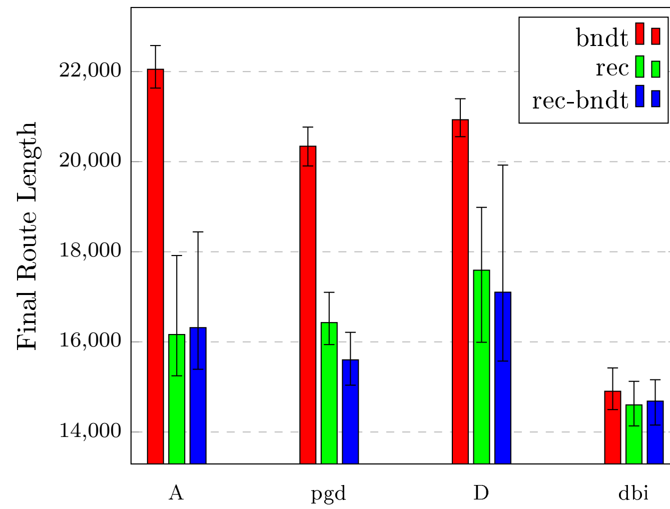


(b) ftv170

Figure 7.15: Search approach performance in ATSP ftv44 and ftv170
Shortest route length found as a function of number of configurations tested by the search approach. The lines represent the median values aggregated over 75 tuning runs.

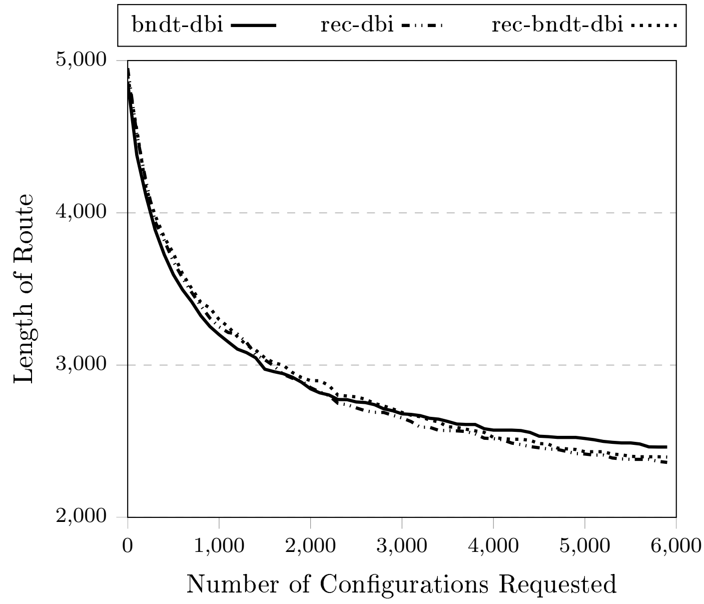


(a) ftv44

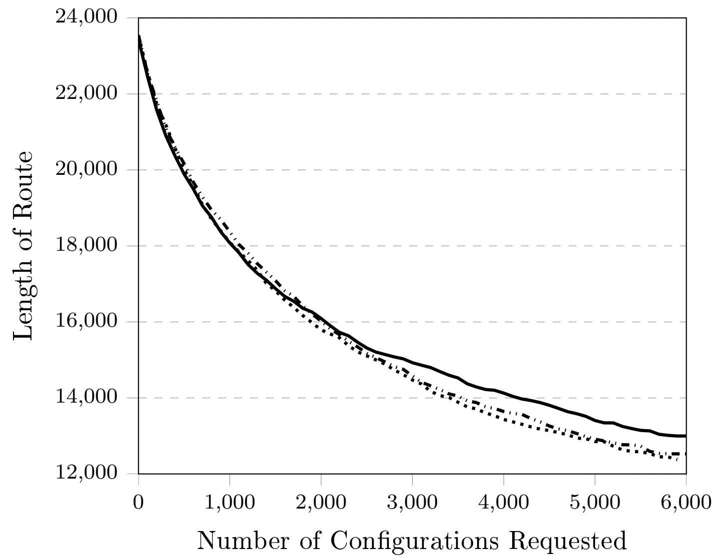


(b) ftv170

Figure 7.16: End of autotuning best route lengths for the ATSP ftv44 and ftv170 benchmarks
Best route lengths found by search approaches after 3000 configurations tested. The search approaches are grouped by initial *technique set*. Values shown are medians over 75 tuning runs.



(a) ftv44



(b) ftv170

Figure 7.17: Database initialized search approaches in the ATSP ftv44 and ftv170 benchmarks
Shortest route length found as a function of number of configurations tested by the search approach. The lines represent the median values aggregated over 75 tuning runs.

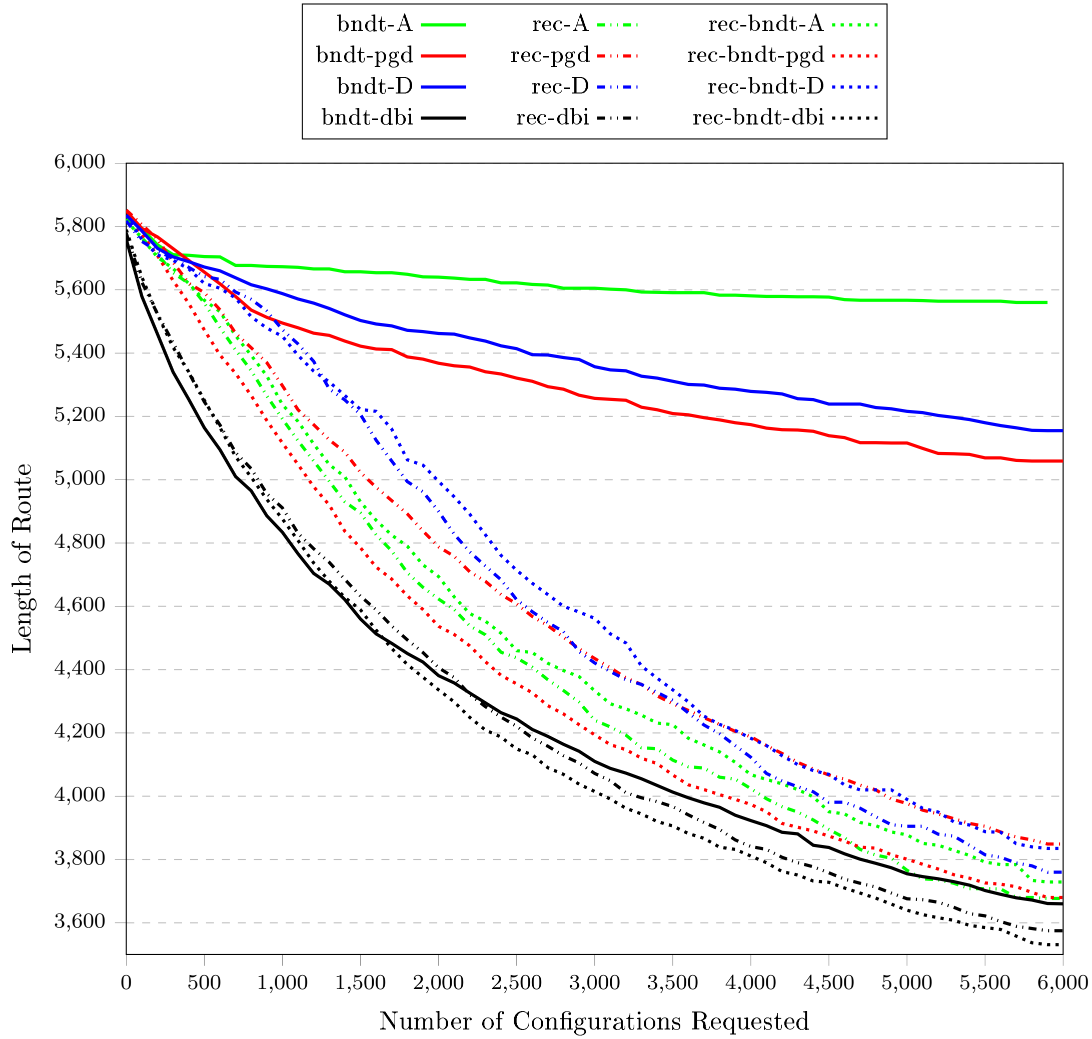


Figure 7.18: Search approach performance in the ATSP rbg323 representation
Shortest route length found as a function of number of configurations tested. The lines represent the median values aggregated over 75 tuning runs.

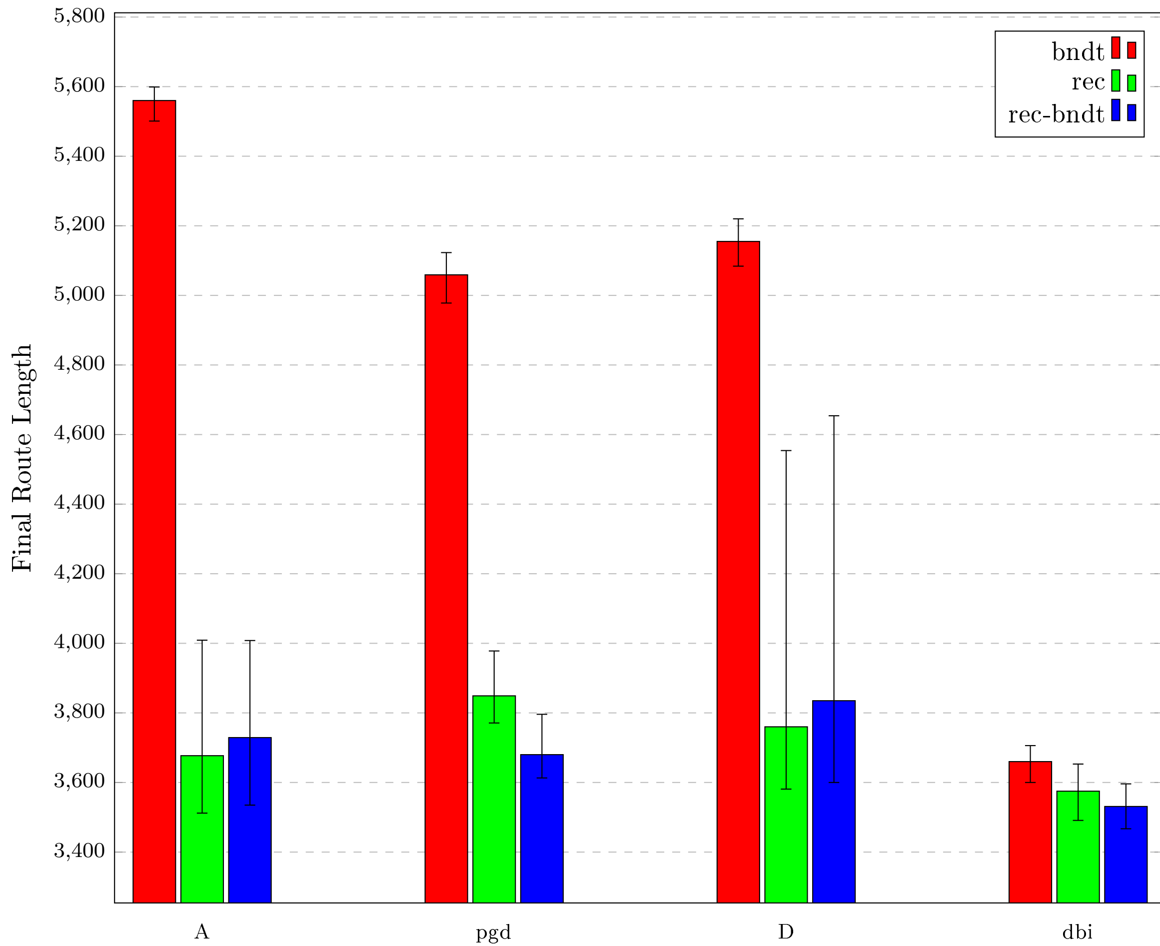


Figure 7.19: End of autotuning best route lengths for the ATSP rbg323 benchmark
Best route lengths found by search approaches after 6000 configurations tested. The search approaches are grouped by initial *technique set*. Values shown are medians over 75 tuning runs.

Performance results on the rbg323 ATSP benchmark in Figures 7.18 and 7.19 again show how technique recommendation allows for fine tuning of the *technique set* and better autotuning performance over the course of the tuning run. Regardless of the initial *technique set*, all of the rec and rec-bndt search approaches eventually exhibit similar autotuning performance to bndt-dbi, with rec-A, rec-bndt-dbi, and rec-dbi exceeding bndt-dbi's performance within the scope of the autotuning sessions. The consistently strong performance of the rec and rec-bandit relative to the corresponding bndt across all initial *technique sets* demonstrates the robustness that recommendation brings.

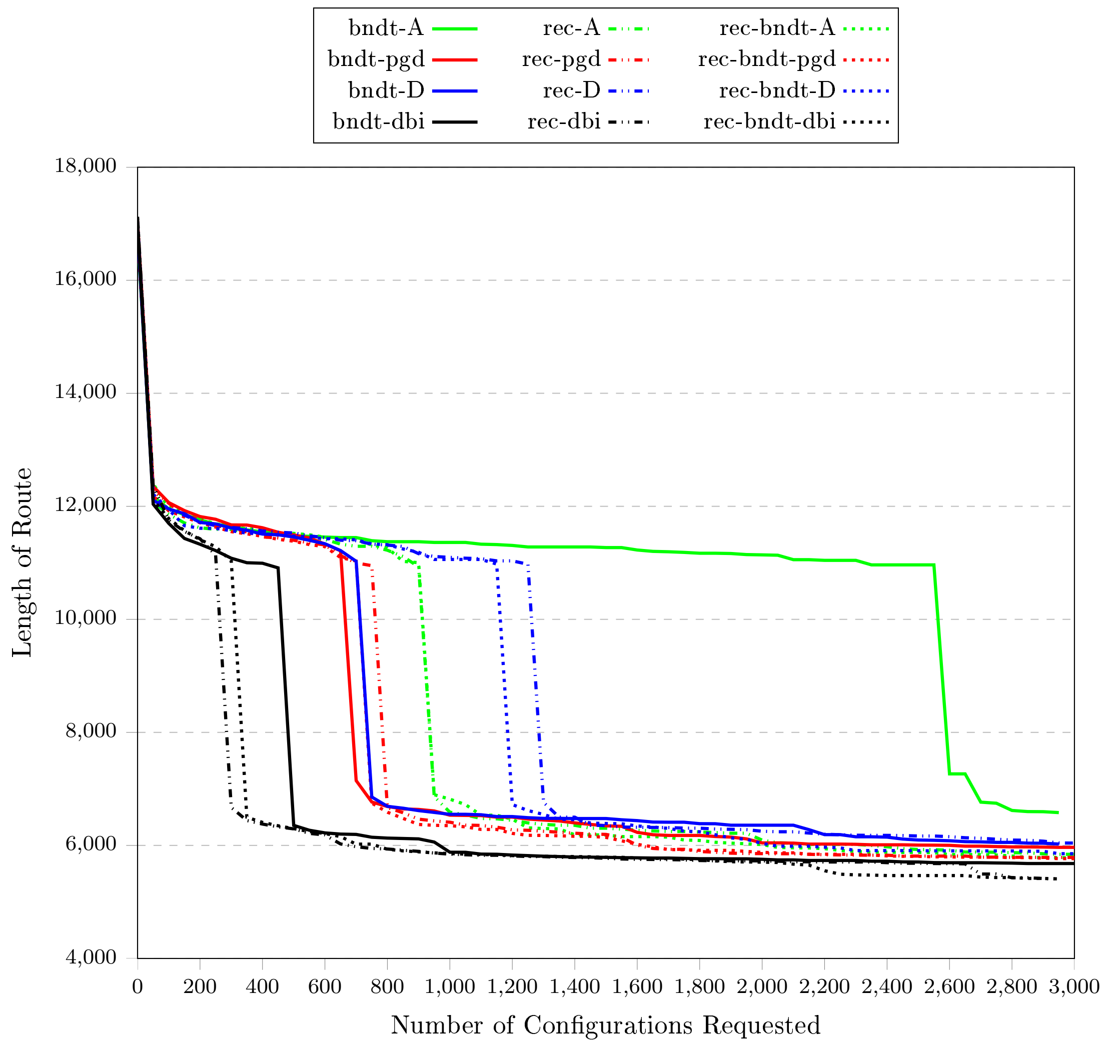


Figure 7.20: Search approach performance in the ATSP p43 representation
Shortest route length found as a function of number of configurations tested. The lines represent the median values aggregated over 75 tuning runs.

Finally, Figure 7.20 shows search approach performances on the p43 ATSP benchmark. In the p43 benchmark, certain problem cities have prohibitively high distances to all but a few of the other cities. This results in the step-like autotuning profile with large, sharp drops in route length when a particular

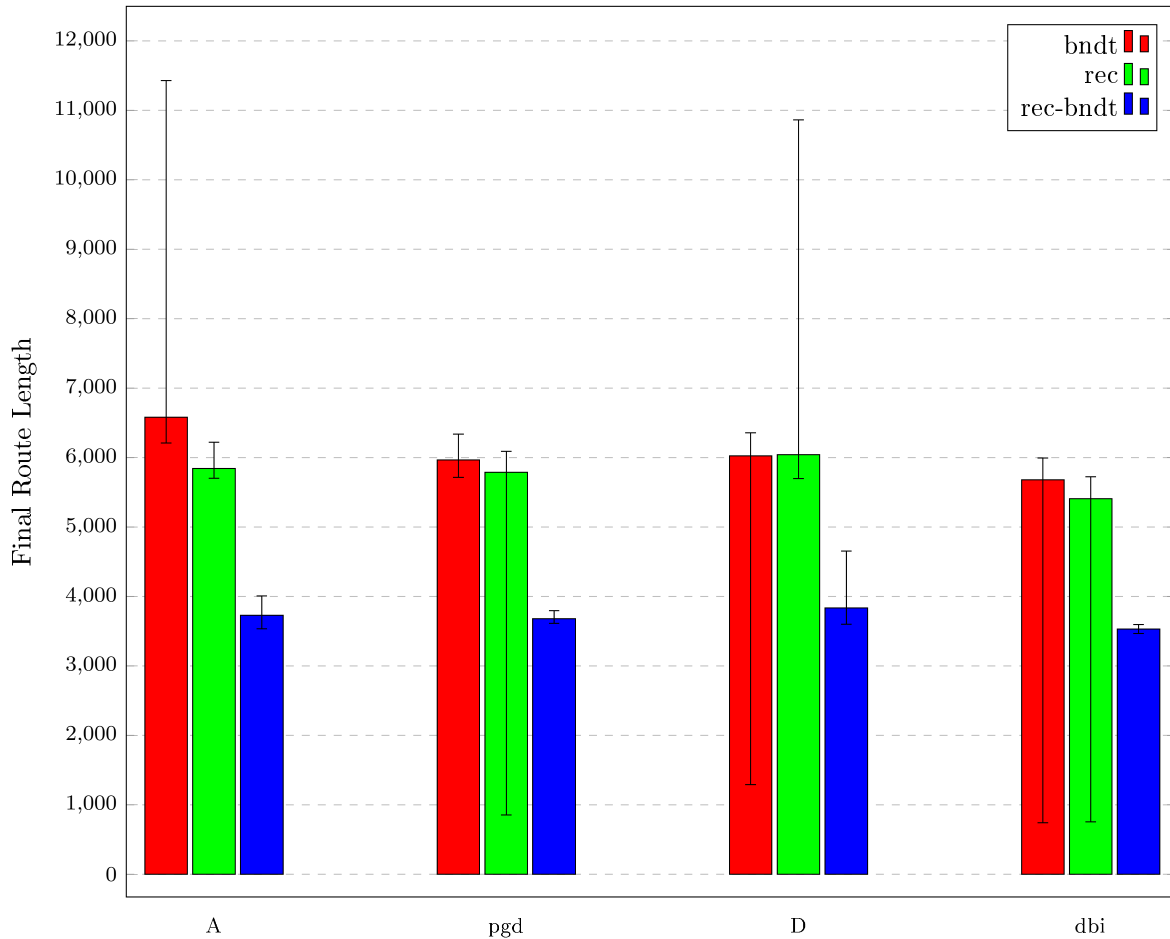


Figure 7.21: End of autotuning best route lengths for the ATSP p43 benchmark
Best route lengths found by search approaches after 3000 configurations tested. The search approaches are grouped by initial *technique set*. Values shown are medians over 75 tuning runs.

sub-permutation yielding a good path through a problem city is found.

As seen in Figure 7.21, the rec-bndt search approaches find better final routes than their bndt counterparts. Again, search approaches with the dbi initial *technique set* yield the best autotuning performance, while bndt-A yields the worst performance. Among the database initialized search approaches, the rec and rec-bndt outperform the bndt, showing the advantages of recommendation.

However, for initial *technique set* D, the bndt out-performs both the rec and rec-bndt in terms of convergence. While this initially appears to be a flaw in the recommendation process itself, this behaviour may actually stem from our use of new bests per desired result as a metric for search affinity.

The heightened convergence speed of the bndt is largely due to obtaining large decreases in route length from discovering the good path through a problem city faster. On the other hand, the rec and rec-bndt see a steadier decrease in route length first, which results in more new bests discovered. Thus, the recommender system has in fact succeeded in it's goal of recommending techniques with better search affinity as defined by our heuristic.

Unfortunately, these techniques showed poorer real autotuning performance than those in initial *technique set* D. While those techniques discovered fewer new bests, they ones they did find resulted in gains of a larger magnitude. This discrepancy between search affinity and true autotuning search performance could be resolved by altering the metric for calculating search affinity to better account not only the number of new bests, but also their magnitude of improvement.

Overall though, rec-bndt-dbi, the transfer learning search approach utilizing the DBI recommendation bandit meta-technique, completely outperforms bndt-A, which is OpenTuner's default in-situ learning search approach. The best route lengths found, relative to the best known route length, for the two search approaches are summarized in Table 7.3 for each of the TSP benchmarks we tested.

7.3 Key Findings

We summarize some of the key conclusions from our experiments:

Search approaches using recommendation are more robust w.r.t. the initial *technique set*

> The performance of the standard bndt search approaches was highly dependent on the initial *technique set*. The pgd *technique set* showed the overall best performance, however we noticed that choosing the wrong *technique set* resulted in poor asymptotic autotuning performance. In fact, the worst search approach for all benchmarks was a bndt. In contrast, the search approaches using recommendation with the same initial *technique set* either performed significantly better than, or on par with, their bndt counterpart.

Benchmark	Cfgs Tested	bndt-A	rec-bndt-dbi	Improvement
att48	3000	88536	40348	2.2x
dantzig42	3000	1264	338	3.7x
ftv44	3000	2685	783	3.4x
ftv170	3000	19296	9621	2.0x
p43	3000	5942	4769	1.2x
rbg323	6000	4234	2205	1.9x
average	-	-	-	2.4x

Table 7.3: Relative performance of bndt-A and rec-bndt-dbi on TSP benchmarks
For each benchmark, we give the total number of configurations evaluated in the tuning runs. This is followed by distance from the best known route length to the best route lengths found by the default OpenTuner search approach (bndt-A) and the DBI recommendation bandit meta-technique (rec-bndt-dbi) at the end of autotuning. Best known route lengths are given in Table 7.2. We use the median value aggregated over 75 tuning runs.

The new rec-bndt-dbi outperforms the old OpenTuner default bndt-A

The rec-bndt-dbi using the new DBI recommendation bandit meta-technique was highly robust, consistently showing some of the strongest autotuning performance in all benchmarks. On the other hand, although the bndt-A performed well in the Petabricks and Halide domains, it showed the worst performance out of all tested search approaches in the TSP domain. Within the TSP domain, the rec-bndt-dbi achieved up to a 3.7x performance improvement over the bndt-A.

Our metric for measuring the search affinity of a technique may need reconsideration

The current method of taking the new bests per desired result as a metric for search affinity ignores the quality of those new bests. This could be creating a gap between true autotuning performance and values given by our metric, resulting in poor recommendation performance for initial *technique set* D in the p43 ATSP benchmark.

Chapter 8

Future Work

8.1 Expanding Recommendation in DeepTuner

The work in this thesis represents a first step into search approach recommendation for general autotuning frameworks. After our demonstration of recommendation at the technique level, the logical next step is to explore recommendation at other points in the search approach hierarchy, such as at the meta-technique or operator level.

An additional direction may be to investigate search approach recommendation that takes into account the "stage" of autotuning. Doing so may better account for differences in search affinity for techniques across the earlier, middle, and later stages of autotuning. For example, techniques can more easily discover new bests in early stages of autotuning, while later stages of autotuning may require more variable techniques that are normally suboptimal to discover new bests.

8.2 Search Affinity Metrics

While taking new bests per desired result gives a quick and simple metric for search affinity, our results showed that this naive approach may not be sufficient. Because only quantity of new bests is considered and their quality is ignored, the current metric does not necessarily translate into true search affinity. Developing robust, problem definition agnostic metrics for search affinity that are more accurate can lead to better metrics for similarity between representations. This would help improve the final performance of recommendation.

8.3 Other Modes for Transfer Learning

Although we choose to utilize search approach recommendation in this work to enable transfer learning in search approaches, there are many alternate methods for transfer learning that could be explored. For instance, just as in-situ learning in OpenTuner allowed techniques to share knowledge about good configurations, we could apply transfer learning directly to the configuration space and allow for recommendation of potentially good configurations across tuning runs.

8.4 Addressing Privacy Concerns

While the cross-session database is a powerful tool containing a wealth of historical tuning run data, maintaining a globally shared database can often raise privacy issues. Currently, the cross-session database records many specifics about each tuning run. By intelligently aggregating this information across many users and tuning runs, we can reduce the exposure of information about specific individuals.

8.5 Developing New Parameter Types and Operators

We believe that one reason for search approach recommendation's larger performance gains on TSP is due to the wide array of operators available for the permutation parameter and the fact that the TSP problem is nicely represented as a permutation. In other words, there are many good techniques to recommend. Having more operators results in a larger variety of techniques, a subset of which are potentially much better at searching a given representation - conditions that call for recommendation. On the other hand, well-structured parameter types for a problem mean more opportunities to exploit that structure. This leads to better operators, which again increases potential number of good techniques to recommend.

Thus, development of new parameters (such as a tree-like parameter) better suited for various problems or creation of new operators can greatly impact autotuning performance.

Chapter 9

Conclusion

By introducing technique composition in OpenTuner, we greatly increased the variety of different techniques that can be expressed in OpenTuner. However, this increase in the number of potentially good techniques also led to an explosion in the number of suboptimal search approaches, making the previous method of finding a good in-situ learning search approach manually or through anecdotal evidence even less feasible.

To remedy the situation, we introduce transfer learning search approaches created from the recommendation bandit meta-technique and the DBI recommendation bandit meta-technique that take advantage of DeepTuner's cross-session database. We demonstrate the robustness of these search approaches on nine representations over three domains where the new search approaches using recommendation achieve performance on-par with or greater than that of their corresponding traditional in-situ learning search approaches in OpenTuner.

In particular, the DBI recommendation bandit meta-technique performs on par with both the default and an anecdotally good in-situ learning search approach for the Halide and Petabricks domains. Within the TSP domain, the DBI recommendation bandit meta-technique vastly outperforms the other two, achieving a 2.4x average performance gain over the OpenTuner's default search approach for tested benchmarks.

In addition to its robustness, a search approach using the DBI recommendation bandit meta-technique does not require specification of an initial set of techniques before autotuning begins, unlike with a traditional AUC bandit meta-technique based in-situ learning search approach. Thus, our new DBI recommendation bandit meta-technique not only provides better, more robust autotuning performance, but it also streamlines the autotuning process for the user by removing the need for manually selecting a good search approach, as automatically performed dynamically during the autotuning session.

The current DeepTuner and DBI recommendation bandit meta-technique represent a proof of concept for

the new ideas of transfer learning search approaches and search approach recommendation in OpenTuner. It is our hope that future work exploring these concepts will continue to yield advances in the performance of program autotuner frameworks, allowing users of OpenTuner to experience even greater benefits.

Bibliography

[1] Tsplib. Available at http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/ Accessed: 2015-04-16.

[2] Differential evolution implementation, 2015. available at http://cci.lbl.gov/cctbx_sources/scitbx/differential_evolution.py, accessed: 2015-06-16.

[3] Rehab F Abdel-Kader. Hybrid discrete pso with ga operators for efficient qos-multicast routing. *Ain Shams Engineering Journal*, 2(1):21–31, 2011.

[4] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. *PetaBricks: a language and compiler for algorithmic choice*, volume 44. ACM, 2009.

[5] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation Techniques*, Edmonton, Canada, August 2014.

[6] Jee W Choi, Amik Singh, and Richard W Vuduc. Model-driven autotuning of sparse matrix-vector multiply on gpus. In *ACM Sigplan Notices*, volume 45, pages 115–126. ACM, 2010.

[7] Matthias Christen, Olaf Schenk, and Helmar Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 676–687. IEEE, 2011.

[8] Lawrence Davis. Applying adaptive algorithms to epistatic domains. In *IJCAI*, volume 85, pages 162–164, 1985.

[9] Kusum Deep and Hadush Mebrahtu Adane. New variations of order crossover for travelling salesman problem. *IJCOPI*, 2(1):2–13, 2011.

[10] Will Drevo. Delphi: A distributed multi-algorithm, multi-user, self optimizing machine learning system. Master's thesis, Massachusetts Institute of Technology, 2014.

[11] Álvaro Fialho, Luis Da Costa, Marc Schoenauer, and Michèle Sebag. Analyzing bandit-based adaptive operator selection mechanisms. *Annals of Mathematics and Artificial Intelligence*, 60(1-2):25–64, October 2010.

[12] Álvaro Fialho, Raymond Ros, Marc Schoenauer, and Michèle Sebag. Comparison-based adaptive strategy selection with bandits in differential evolution. In *Proceedings of the 11th International Conference on Parallel Problem Solving from Nature: Part I*, PPSN'10, pages 194–203, Berlin, Heidelberg, 2010. Springer-Verlag.

[13] Matteo Frigo and Steven G Johnson. The design and implementation of fftw3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[14] David E Goldberg and Robert Lingle. Alleles, loci, and the traveling salesman problem. In *Proceedings of the first international conference on genetic algorithms and their applications*, pages 154–159. Lawrence Erlbaum Associates, Publishers, 1985.

[15] Herbert Jordan, Peter Thoman, Juan J Durillo, Sara Pellegrini, Philipp Gschwandtner, Thomas Fahringer, and Hans Moritsch. A multi-objective auto-tuning framework for parallel codes. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–12. IEEE, 2012.

[16] Chia-Feng Juang. A hybrid of genetic algorithm and particle swarm optimization for recurrent network design. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 34(2):997–1006, 2004.

[17] Shoaib Kamil, Cy Chan, Leonid Oliker, John Shalf, and Samuel Williams. An auto-tuning framework for parallel multicore stencil computations. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[18] Shoaib A Kamil. *Productive high performance parallel programming with auto-tuned domain-specific embedded languages*. PhD thesis, January 2013.

[19] Manuel Lozano, Francisco Herrera, Natalio Krasnogor, and Daniel Molina. Real-coded memetic algorithms with crossover hill-climbing. *Evolutionary computation*, 12(3):273–302, 2004.

[20] Thibaut Lutz, Christian Fensch, and Murray Cole. Partans: An autotuning framework for stencil computation on multi-gpu systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):59, 2013.

[21] Tadahiko Murata, Hisao Ishibuchi, and Hideo Tanaka. Genetic algorithms for flowshop scheduling problems. *Computers & Industrial Engineering*, 30(4):1061–1071, 1996.

[22] Maciej Pacula, Jason Ansel, Saman Amarasinghe, and Una-May O'Reilly. Hyperparameter tuning in bandit-based adaptive operator selection. In *Applications of Evolutionary Computation*, pages 73–82. Springer, 2012.

[23] SG Ponnambalam and Mohan Reddy. A ga-sa multiobjective hybrid search algorithm for integrating lot sizing and sequencing in flow-line scheduling. *The International Journal of Advanced Manufacturing Technology*, 21(2):126–137, 2003.

[24] Markus Püschel, José MF Moura, Bryan Singer, Jianxin Xiong, Jeremy Johnson, David Padua, Manuela Veloso, and Robert W Johnson. Spiral: A generator for platform-adapted libraries of signal processing alogorithms. *International Journal of High Performance Computing Applications*, 18(1):21–45, 2004.

[25] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

[26] XH Shi, YC Liang, HP Lee, C Lu, and LM Wang. An improved ga and a novel pso-ga-based hybrid algorithm. *Information Processing Letters*, 93(5):255–261, 2005.

[27] Mark W Stephenson. *Automating the Construction of Compiler Heuristics Using Machine Learning*. PhD thesis, Massachusetts Institute of Technology, 2006.

[28] Cristian Ţăpuş, I-Hsin Chung, Jeffrey K Hollingsworth, et al. Active harmony: Towards automated performance tuning. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–11. IEEE Computer Society Press, 2002.

[29] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffrey K Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE, 2009.

[30] Richard Vuduc, James W Demmel, and Katherine A Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *Journal of Physics: Conference Series*, volume 16, page 521. IOP Publishing, 2005.

[31] Hongfeng Wang, Dingwei Wang, and Shengxiang Yang. A memetic algorithm with adaptive hill climbing strategy for dynamic optimization problems. *Soft Computing*, 13(8-9):763–780, 2009.

[32] R Clint Whaley and Jack J Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–27. IEEE Computer Society, 1998.

[33] Darrell Whitley, Doug Hains, and Adele Howe. Tunneling between optima: partition crossover for the traveling salesman problem. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 915–922. ACM, 2009.

[34] Wen-Jun Zhang, Xiao-Feng Xie, et al. Depso: hybrid particle swarm with differential evolution operator. In *IEEE International Conference on Systems Man and Cybernetics*, volume 4, pages 3816–3821, 2003.

Appendix A

Appendix A: Definitions and Terminology

We note that the terminology used in this thesis may differ from the terminology used in previous works. This is partially due to an effort to disambiguate the old terminology.

A.1 Basic Autotuning Terminology

We frame the autotuning problem as a search problem over the space of the variables being tuned. This is reflected in the terminology used for discussing autotuning in OpenTuner.

parameter

the dimensions, or variables, of the autotuning search space. In OpenTuner, this is any instance of Parameter class representing a variable in the autotuning space (e.g. permutation parameter, boolean parameter, ...).

parameters in the representation

the set of parameters the user chooses to define the dimensions of the autotuning search space as.

configuration

a concrete assignment of values to a set of parameters. These make up the "points" in autotuning search space.

objective function

the method used to evaluate the fitness of a configuration. Note that this encapsulates the process of

interpreting the "meaning" of parameters in the representation.

tuning run

a single autotuning session

A.2 Autotuner Specification

We equate the process of implementing an autotuner in OpenTuner to the process of defining the autotuning problem in a hierarchical way. See Section 3.1 for more detail about the hierarchical structure of a *problem definition*.

representation

the autotuning search space in a tuning run. A representation consists of the choice of *parameters in the representation* to use as the search space and the *objective function* used to interpret and score a configuration.

For example, if we wished to autotune the GCC compilation flags that can be turned on or off to minimize the execution time of a program, we could use a simple representation consisting of a boolean parameter for each compiler flag. The parameters in the representation would be the set of boolean parameters. The objective function would consist of

1. compiling the program with all flags whose corresponding boolean parameters are true turned on
2. evaluating the execution time of the compiled program.

problem

the specific program or use case that a representation maps to. The problem is essentially the actual thing being tuned by the autotuner during a tuning run.

For instance, in the above example about a representation for autotuning GCC compilation flags, the problem would be the specific programs or application being compiled, such as a program performing a matrix multiply, whose execution time we are trying to minimize.

domain

the overarching project or application that an implemented autotuner operates over. A domain consists of a set of related problems which often have similar representations using the same objective function.

In the above examples, the domain would be selecting GCC compilation flags to minimize overall execution time of a C program.

problem definition

the complete specification of an autotuner and autotuning problem in the OpenTuner framework, from a domain down to the representation.

A.3 Search Approach

We consider the autotuning process during a single session as the execution of some user-specified *search approach* that sequentially selects configurations to evaluate.

technique

a standalone search algorithm for exploring the autotuning search space of configurations. A technique selects the next configuration to evaluate based only on the fitness of previously evaluated configurations. A technique is standalone in the sense that it can operate without knowledge of other techniques.

Techniques can be decomposed into a technique base and a set of operators.

technique base

the algorithmic component of a technique that determines which configuration/configurations the next configuration to be evaluated is/are generated from. In other words, the technique base is how a technique maintains and selects from a population of candidate configurations that will be combined or mutated to generate the next configuration. This is equivalent to parent selection in a genetic algorithm based technique.

For instance, consider a normal greedy mutation technique that always selects the global best configuration and mutates a subset of its parameters according to a normal distribution. The technique base would only capture the fact that the technique of selects the global best configuration. The fact that this configuration then undergoes a random mutation according to a normal distribution is is unrelated.

operator

an algorithm used to generate a new parameter value from a set of parameter values. Every operator is associated with a specific parameter type. We note that picking a set of operators for a technique, one for each parameter type in a configuration, defines how a set of candidate configurations will be combined into a final configuration to be tested.

composable technique

a technique where the set of operators can be changed arbitrarily while keeping the same technique base

to generate a new, coherent technique. Most techniques can be thought of as composable techniques, except the case where specific operators are deeply entrenched in the technique base's process for generating candidate configurations.

meta-technique

an algorithm that explores the autotuning search space by repeatedly choosing other techniques to generate the next configuration with. A meta-technique can be thought of as a selection algorithm over some set of underlying techniques that are used to actually conduct the exploration.

hyper-parameter

an input key-word argument whose value modifies the behaviour of any algorithmic component of a search approach. For example, we could have a meta-technique hyper-parameter, a technique hyper-parameter, or a operator hyper-parameter. A hyper-parameter must be a literal such as a boolean, integer, float, or string.

The value of a hyper-parameter is set when the technique or meta-technique is initialized.

search approach

the overarching search process used to search an autotuning search space. In OpenTuner, this is the fully specified algorithm that can immediately be used to explore the autotuning search space defined by the representation. A search approach interacts with the search space of configurations during the autotuning process by making requests for configurations to be evaluated (desired results). The resulting evaluations are used to inform future requests. A search approach consists of a meta-technique bundled with techniques where all hyper-parameters or operator choices have been fixed. We note that we can have the "null" meta-technique, i.e. an individual technique is also a search approach. The search approach used for a tuning run is specified by the user at the start of tuning through the "–technique" command line argument.

A naive search approach has no learning aspect outside of the individual technique. In other words, this is the individual technique.

An in-situ learning search approach performs learning on top of the individual techniques using data gained during its current tuning run.

A transfer learning search approach performs learning on top of the individual techniques using data gained from external tuning runs.

A.4 Search and Evaluation

OpenTuner abstracts the process of searching the space of configurations from the process of evaluating configurations.

desired result

a configuration that is requested to be evaluated by a particular search approach or technique.

result

the evaluation of a desired result according to the objective function.

new best

a resultthat is the global best found during a tuning run at the time of its evaluation.

measurement driver

the module of OpenTuner that handles measuring desired results to produce results. This is a thin wrapper around the user defined objective function.

search driver

the module of OpenTuner that handles generating desired results to explore the search space. Encapsulates the problem definition and search approach.

results database

a database used by the measurement driver and the search driver to communicate during a single tuning run. It tracks desired results requested and results collected during a tuning run, along with metadata about the problem definition and search approach for the tuning run.

A.5 Recommender System

In this work, we introduce DeepTuner which sits above all individual OpenTuner tuning runs to provide insight cross-tuning runs.

search affinity

a measure of how well a search approach, or any individual subcomponent of a search approach such as a technique interacts with a specific problem definition. In this thesis, we use the metric :

$$\frac{\text{number of new bests yielded}}{\text{number of desired results requested}}$$

For example, if a technique requested 10 desired results and 5 of these configurations turned out to be new bests at the time of evaluation, then the search affinity of the technique would be 0.5. This is essentially a measure of how efficient the technique is at choosing new globally optimal configurations to evaluate.

cross-session database

a database used by DeepTuner containing uploaded data about past tuning runs. It stores values of the search affinity at regular intervals of techniques and the meta-technique for a tuning run, along with meta data about the problem definition. The cross-session databasecan also tracks aggregate values of search affinityover multiple tuning runs.