Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2013-026 November 1, 2013

OpenTuner: An Extensible Framework for Program Autotuning

Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Una-May O Reilly, and Saman Amarasinghe

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

OpenTuner: An Extensible Framework for Program Autotuning

Jason Ansel Shoaib Kamil Kalyan Veeramachaneni Una-May O'Reilly Saman Amarasinghe

Massachusetts Institute of Technology

{jansel, skamil, kalyan, unamay, saman}@csail.mit.edu

Abstract

Program autotuning has been shown to achieve better or more portable performance in a number of domains. However, autotuners themselves are rarely portable between projects, for a number of reasons: using a domain-informed search space representation is critical to achieving good results; search spaces can be intractably large and require advanced machine learning techniques; and the landscape of search spaces can vary greatly between different problems, sometimes requiring domain specific search techniques to explore efficiently.

This paper introduces OpenTuner, a new open source framework for building domain-specific multi-objective program autotuners. OpenTuner supports fully-customizable configuration representations, an extensible technique representation to allow for domain-specific techniques, and an easy to use interface for communicating with the program to be autotuned. A key capability inside OpenTuner is the use of ensembles of disparate search techniques simultaneously; techniques that perform well will dynamically be allocated a larger proportion of tests. We demonstrate the efficacy and generality of OpenTuner by building autotuners for 6 distinct projects and 14 total benchmarks, showing speedups over prior techniques of these projects of up to $2.8\times$ with little programmer effort.

OpenTuner can be downloaded from:
`http://opentuner.org/`

1. Introduction

Program autotuning is increasingly being used in domains such as high performance computing and graphics to optimize programs. Program autotuning augments traditional human-guided optimization by offloading some or all of the search for an optimal program implementation to an automated search technique. Rather than optimizing a program directly, the programmer expresses a search space of possible implementations and optimizations. Autotuning can often make the optimization process more efficient as autotuners are able to search larger spaces than is possible by hand. Autotuning also provides performance portability, as the autotuning process can easily be re-run on new machines which require different sets of optimizations. Finally, multi-objective autotuning can be used to trade off between performance and accuracy, or other criteria such as energy consumption and memory usage, and provide programs which meet given performance or quality of service targets.

While the practice of autotuning has increased in popularity, autotuners themselves often remain relatively simple and project specific. There are three main challenges which make the development of autotuning frameworks difficult.

The first challenge is using the right configuration representation for the problem. Configurations can contain parameters that vary from a single integer for a block size to a much more complex type such as an expression tree representing a set of instructions. The creator of the autotuner must find ways to represent their complex domain-specific data structures and constraints. When these data structures are naively mapped to simpler representations, such as a point in high dimensional space, locality information is lost which makes the search problem much more difficult. Picking the right representation for the search space is critical to having an effective autotuner. To date, all autotuners that have used a representation other than the simplest ones have had custom project-specific representations.

The second challenge is the size of the valid configuration space. While some prior autotuners have worked hard to prune the configuration space, we have found that for many problems excessive search space pruning will miss out on non-intuitive good configurations. We believe providing all the valid configurations of these search spaces is better than artificially constraining search spaces and possibly missing optimal solutions. Search spaces can be very large, up to 10^{3600} possible configurations for one of our benchmarks. Full exhaustive search of such a space will not complete in human lifetimes! Thus, intelligent machine learning techniques are required to seek out a good result with a small number of experiments.

The third challenge is the landscape of the configuration space. If the configuration space is a monotonic function, a search technique biased towards this type of search space (such as a hill climber) will be able to find the optimal configuration. If the search space is discontinuous and haphazard an evolution algorithm may perform better. However, in practice search spaces are much more complex, with discon-

tinuities, high dimensionality, plateaus, hills with some of the configuration parameters strongly coupled and some others independent from each other. A search technique that is optimal in one type of configuration space may fail to locate an adequate configuration in another. It is difficult to provide a robust system that performs well in a variety of situations. Additionally, many application domains will have domain-specific search techniques (such as scheduling or blocking heuristics) which may be critical to finding an optimal solution efficiently. This has caused most prior autotuners to use customized search techniques tailored to their specific problem. This requires machine learning expertise in addition to the individual domain expertise to build an autotuner for a system. We believe that this is one of the main reasons that, while autotuners are recognized as critical for performance optimization, they have not seen commodity adoption.

In this paper we present OpenTuner, a new framework for building domain-specific program autotuners. OpenTuner features an extensible configuration and technique representation able to support complex and user-defined data types and custom search heuristics. It contains a library of predefined data types and search techniques to make it easy to setup a new project. Thus, OpenTuner solves the custom configuration problem by providing not only a library of data types that will be sufficient for most projects, but also extensible data types that can be used to support more complex domain specific representations when needed.

A core concept in OpenTuner is the use of *ensembles* of search techniques. Many search techniques (both built in and user-defined) are run at the same time, each testing candidate configurations. Techniques which perform well by finding better configurations are allocated larger budgets of tests to run, while techniques which perform poorly are allocated fewer tests or disabled entirely. Techniques are able to share results using a common results database to constructively help each other in finding an optimal solution. To allocate tests between techniques we use an optimal solution to the *multi-armed bandit* problem using area under the curve credit assignment. Ensembles of techniques solve the large and complex search space problem by providing both a robust solutions to many types of large search spaces and a way to seamlessly incorporate domain specific search techniques.

1.1 Contributions

This paper makes the following contributions:

- To the best of our knowledge, OpenTuner is the first to introduce a general framework to describe complex search spaces for program autotuning.
- OpenTuner introduces the concept of ensembles of search techniques to program autotuning, which allow many search techniques to work together to find an optimal solution.
- OpenTuner provides more sophisticated search techniques than typical program auotuners. This enables expanded uses of program autotuning to solve more complex search problems and pushes the state of the art forward in program autotuning in a way that can easily be adopted by other projects.
- We demonstrate the versatility of our framework by building autotuners for 6 distinct projects and demonstrate the effectiveness of the system with 14 total benchmarks, showing speedups over existing techniques of up to $2.8\times$.
- We show that OpenTuner is able to succeed both in massively large search spaces, exceeding 10^{3600} possible configurations in size, and in smaller search spaces using less than 2% of the tests required for exhaustive search.

2. Related Work

Package	Domain	Search Method
Active Harmony [30]	Runtime System	Nelder-Mead
ATLAS [33]	Dense Linear Algebra	Exhaustive
FFTW [14]	Fast Fourier Transform	Exhaustive/Dynamic Prog.
Insieme [19]	Compiler	Differential Evolution
OSKI [32]	Sparse Linear Algebra	Exhaustive+Heuristic
PATUS [9]	Stencil Computations	Nelder-Mead or Evolutionary
PetaBricks [4]	Programming Language	Bottom-up Evolutionary
Sepya [21]	Stencil Computations	Random-Restart Gradient Ascent
SPIRAL [27]	DSP Algorithms	Pareto Active Learning

Figure 1. Summary of selected related projects using autotuning

A number of offline empirical autotuning frameworks have been developed for building efficient, portable libraries in specific domains; selected projects and techniques used are summarized in Figure 1. ATLAS [33] utilizes empirical autotuning to produce an optimized matrix multiply routine. FFTW [14] uses empirical autotuning to combine solvers for FFTs. Other autotuning systems include SPIRAL [27] for digital signal processing PATUS [9] and Sepya [21] for stencil computations, and OSKI [32] for sparse matrix kernels.

The area of iterative compilation contains many projects that use different machine learning techniques to optimize lower level compiler optimizations [1, 2, 15, 25]. These projects change both the order that compiler passes are applied and the types of passes that are applied.

In the dynamic autotuning space, there have been a number of systems developed [5, 6, 8, 16, 18, 22] that focus on creating applications that can monitor and automatically tune themselves to optimize a particular objective. Many of these systems employ a control systems based autotuner that operates on a linear model of the application being tuned. For example, PowerDial [18] converts static configuration parameters that already exist in a program into dynamic knobs that can be tuned at runtime, with the goal of trading QoS guarantees for meeting performance and power usage goals. The system uses an offline learning stage to construct a linear model of the choice configuration space which

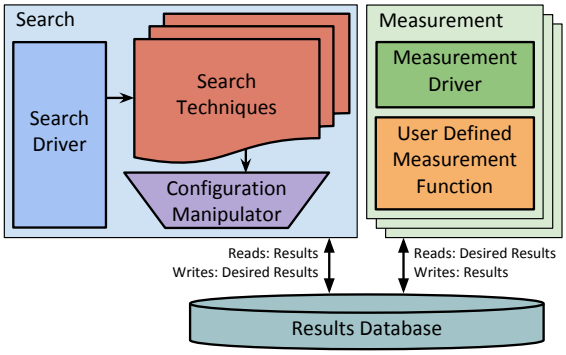


Figure 2. Overview of the major components in the OpenTuner framework.

can be subsequently tuned using a linear control system. The system employs the heartbeat framework [17] to provide feedback to the control system. A similar technique is employed in [16], where a simpler heuristic-based controller dynamically adjusts the degree of loop perforation performed on a target application to trade QoS for performance.

3. The OpenTuner Framework

Our terminology reflects that the autotuning problem is cast as a search problem. The search space is made up of *configurations*, which are concrete assignments of a set of *parameters*. Parameters can be *primitive* such as an integer or *complex* such as a permutation of a list. When the performance, output accuracy, or other metrics of a configuration are measured (typically by running it in a domain-specific way), we call this measurement a *result*. *Search techniques* are methods for exploring the search space and make requests for measurement called *desired results*. Search techniques can change configurations using a user-defined *configuration manipulator*, which also includes *parameters* corresponding directly the parameters in the configuration. Some parameters include *manipulators*, which are opaque functions that make stochastic changes to a specific parameter in a configuration.

Figure 2 provides an overview of the major components in OpenTuner. The *search* process includes techniques, which use the user defined configuration manipulator in order to read and write configurations. The *measurement* processes evaluate candidate configurations using a user defined measurement function. These two components communicate exclusively through a *results database* used to record all results collected during the tuning process, as well as the providing ability to perform multiple measurements in parallel.

3.1 OpenTuner Usage

To implement an autotuner with OpenTuner, first, the user must define the search space by creating a *configuration manipulator*. This configuration manipulator includes a set of parameter objects which OpenTuner will search over. Second, the user must define a *run* function which evaluates the fitness of a given configuration in the search space to preduce a result. These must be implemented in a small Python program in order interface with the OpenTuner API.

Figure 3 shows an example of using OpenTuner to search over the space of compiler flags to GCC in order to minimize execution time of the resulting program. In Section 4, we present results on an expanded version of this example which obtains up to 2.8x speedup over `-O3`.

This example tunes three types of flags to GCC. First it choses between the four optimization levels `-O0`, `-O1`, `-O2`, `-O3`. Second, for 176 flags listed on line 8 it decides between turning the flag on (with `-fFLAG`), off (with `-fno-FLAG`), or omitting the flag in order to let default value to take precedence. Including the default value as a choice is not necessary for completeness, but speeds up convergence and results in shorter command lines. Finally, it assigns a bounded integer value to the 145 parameters on line 15 with the `--param NAME=VALUE` command line option.

The method `manipulator` (line 23), is called once at startup and creates a `ConfigurationManipulator` object which defines the search space of GCC flags. All accesses to configurations by search techniques are done through the configuration manipulator. For optimization level, an `IntegerParameter` between 0 and 3 is created. For each flag, a `EnumParameter` is created which can take the values `on`, `off`, and `default`. Finally, for the remaining bounded GCC parameters, an `IntegerParameter` is created with the appropriate range.

The method `run` (line 40), implements the measurement function for configurations. First, the configuration is realized as specific command line to `g++`. Next, this `g++` command line is run to produce an executable, `tmp.bin`, which is then run using `call_program`. `Call_program` is a convince function which runs and measures the execution time of the given program. Finally, a `Result` is constructed and returned, which is a database record type containing many other optional fields such as `time`, `accuracy`, and `energy`. By default OpenTuner minimizes the `time` field, however this can be customized.

3.2 Search Techniques

To provide robust search, OpenTuner includes techniques that can handle many types of search spaces and runs a collection of search techniques at the same time. Techniques which perform well are allocated more tests, while techniques which perform poorly are allocated fewer tests. Techniques share results through the results database, so that improvements made by one technique can benefit other techniques. OpenTuner techniques are meant to be extended. Users can define custom techniques which implement domain-specific heuristics and add them to *ensembles* of pre-defined techniques.

```python
import opentuner
from opentuner import ConfigurationManipulator
from opentuner import EnumParameter
from opentuner import IntegerParameter
from opentuner import MeasurementInterface
from opentuner import Result

GCC_FLAGS = [
  'align-functions', 'align-jumps', 'align-labels',
  'branch-count-reg', 'branch-probabilities',
  # ... (176 total)
]

# (name, min, max)
GCC_PARAMS = [
  ('early-inlining-insns', 0, 1000),
  ('gcse-cost-distance-ratio', 0, 100),
  # ... (145 total)
]

class GccFlagsTuner(MeasurementInterface):

  def manipulator(self):
    """
    Define the search space by creating a
    ConfigurationManipulator
    """
    manipulator = ConfigurationManipulator()
    manipulator.add_parameter(
      IntegerParameter('opt_level', 0, 3))
    for flag in GCC_FLAGS:
      manipulator.add_parameter(
        EnumParameter(flag,
                      ['on', 'off', 'default']))
    for param, min, max in GCC_PARAMS:
      manipulator.add_parameter(
        IntegerParameter(param, min, max))
    return manipulator

  def run(self, desired_result, input, limit):
    """
    Compile and run a given configuration then
    return performance
    """
    cfg = desired_result.configuration.data
    gcc_cmd = 'g++ raytracer.cpp -o ./tmp.bin'
    gcc_cmd += ' -O{0}'.format(cfg['opt_level'])
    for flag in GCC_FLAGS:
      if cfg[flag] == 'on':
        gcc_cmd += ' -f{0}'.format(flag)
      elif cfg[flag] == 'off':
        gcc_cmd += ' -fno-{0}'.format(flag)
    for param, min, max in GCC_PARAMS:
      gcc_cmd += ' --param {0}={1}'.format(
        param, cfg[param])

    compile_result = self.call_program(gcc_cmd)
    assert compile_result['returncode'] == 0
    run_result = self.call_program('./tmp.bin')
    assert run_result['returncode'] == 0
    return Result(time=run_result['time'])

if __name__ == '__main__':
  argparser = opentuner.default_argparser()
  GccFlagsTuner.main(argparser.parse_args())
```

Figure 3. GCC/G++ flags autotuner using OpenTuner.

Ensembles of techniques are created by instantiating a *meta technique*, which is a technique made up of a collection of other techniques. The OpenTuner search driver interacts with a single *root* technique, which is typically a meta technique. When the meta technique gets allocated tests, it incrementally decides how to divide theses tests among its sub-techniques. OpenTuner contains an extensible class hierarchy of techniques and meta techniques, which can be combined together and used in autotuners.

3.2.1 AUC Bandit Meta Technique

In addition to a number of simple meta techniques, such as round robin, OpenTuner's core meta technique used in results is the *multi-armed bandit with sliding window, area under the curve credit assignment* (AUC Bandit) meta technique. A similar technique was used in [24] in the different context of online operator selection. It is based on an optimal solution to the multi-armed bandit problem [12]. The multi-armed bandit problem is the problem of picking levers to pull on a slot machine with many arms each with an unknown payout probability. It encapsulates a fundamental trade-off between *exploitation* (using the best known technique) and *exploration* (estimating the performance of all techniques).

The AUC Bandit meta technique assigns each test to the technique, t, defined by the formula

$$\arg\max_t (AUC_t + C\sqrt{\frac{2\lg|H|}{H_t}})$$

where $|H|$ is the length of the sliding history window, H_t is the number of times the technique has been used in that history window, C is a constant controlling the exploration/exploitation trade-off, and AUC_t is the credit assignment term quantifying the performance of the technique in the sliding window. The second term in the equation is the exploration term which gets smaller the more often a technique is used.

The area under the curve credit assignment mechanism, based on [13], draws a curve by looking at the history for a specific technique and looking only at if a technique yielded a new global best or not. If the technique yielded a new global best, a upward line is draw, otherwise a flat line is drawn. The area under this curve (scaled to a maximum value of 1) is the total credit attributed to the technique. This credit assignment mechanism can be described more precisely by the formula:

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

where V_t is the list of uses of t in the sliding window history. $V_{t,i}$ is 1 if using technique t the ith time in the history resulted in a speedup, otherwise 0.

3.2.2 Other Techniques

OpenTuner includes implementations of the techniques: differential evolution; many variants of Nelder Mead and Torc-

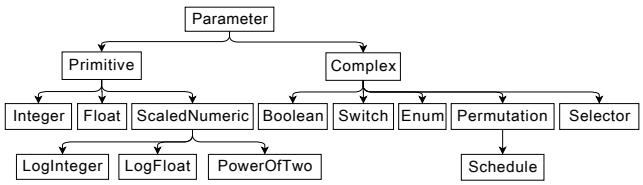


Figure 4. Hierarchy of built in parameter types. User defined types can be added at any point below Primitive or `Complex` in the tree.

zon hillclimbers; a number of evolutionary mutation techniques; pattern search; particle swarm optimization; and random search. OpenTuner also includes a bandit mutation technique which uses the same AUC Bandit method to decide which manipulator function across all parameters to call on the best known configuration. These techniques span a range of strategies and are each biased to perform best in different types of search spaces. They also each contain many settings which can be configured to change their behavior. Each technique has been modified so that with some probability it will use information found by other techniques if other techniques have discovered a better configuration.

The default meta technique, used in results in this paper and meant to be robust, uses an AUC Bandit meta technique to combine greedy mutation, differential evolution, and two hill climber instances.

3.3 Configuration Manipulator

The configuration manipulator provides a layer of abstraction between the search techniques and the raw configuration structure. It is primarily responsible for managing a list of parameter objects, each of which can be used by search techniques to read and write parts of the underlying configuration.

The default implementation of the configuration manipulator uses a fixed list of parameters and stores the configuration as a dictionary from parameter name to parameter-dependant data type. The configuration manipulator can be extended by the user either to change the underlying data structure used for configurations or to support a dynamic list of parameters that is dependant on the configuration instance.

3.3.1 Parameter Types

Figure 4 shows the class hierarchy of built-in parameter types in OpenTuner. Each parameter type is responsible for interfacing between the raw representation of a parameter in the configuration and standardized view of that parameter presented to the search techniques. Parameter types can be extended both to change the underlying representation, and to change the abstraction provided to search techniques to cause a parameter to be search in different ways.

From the viewpoint of search techniques there are two main types of parameters, each of which provides a different abstraction to the search techniques:

Primitive parameters present a view to search techniques of a numeric value with an upper and lower bound. These upper and lower bounds can be dependant on the configuration instance.

The built in parameter types `Float` and `LogFloat` (and similarly `Integer` and `LogInteger`) both have identical representations in the configuration, but present a different view of the underlying value to the search techniques. `Float` is presented directly to to search techniques, while `LogFloat` presents a log scaled view of the underlying parameter to search techniques. To a search technique, halving and doubling a log scaled parameter are changes of equal magnitude. Log scaled variants of parameters are often better for parameters such as block sizes where fixed changes in values have diminishing effects the larger the parameter becomes. `PowerOfTwo` is a commonly used special case, similar to `LogInteger`, where the legal values of the parameter are restricted to powers of two.

Complex parameters present a more opaque view to search techniques. Complex parameters have a variable set of manipulation operators (manipulators) which make stochastic changes to the underlying parameter. These manipulators are arbitrary functions defined on the parameter which can make high level type dependant changes. Complex parameters are meant to be easily extended to add domain specific structures to the search space.

The built in parameter types `Boolean`, `Switch`, and `Enum` could theoretically also be represented as primitive parameters, since they each can be translated directly to a small integer representation. However, in the context of search techniques they make more sense as complex parameters. The reason for this is that for primitive parameters search techniques will attempt to follow gradients. These parameter types are unordered collections of values for which no gradients exist. Thus, the complex parameter abstraction is a more efficient representation to search over.

The `Permutation` parameter type assigns an order to a given list of values and has manipulators which make various types of random changes to the permutation. A `Schedule` parameter is a `Permutation` with a set of dependencies that limit the legal order. Schedules are implemented as a permutation that gets topologically sorted after each change. Finally, a `Selector` parameter is a special type of tree which is used to define a mapping from an integer input to an enumerated value type.

In addition to these primary primitive and complex abstractions for parameter types, there are a number of derived ways that search techniques will interact with parameters in order to more precisely convey intent. These are additional methods on parameter which contain default implementations for both primitive and complex parameter types. These methods can optionally be overridden for specific parameters types to improve search techniques. Parameter types will

work without these methods being overridden, however implementing them can improve results.

As an example, a common operation in many search techniques is to add the difference between configuration A and B to configuration C. This is used both in differential evolution and many hill climbers. Complex parameters have a default implementation of this indent which compares the value of the parameter in the 3 configurations: if $A = B$, then there is no difference and the result is C; similarly, if $B = C$, then A is returned; otherwise a change should be made so random manipulators are called. This works in general, however for individual parameter types there are often better interpretations. For example with permutations, one could calculate the positional movement of each item in the list an calculate a new permutation by applying these movements again.

3.4 Objectives

OpenTuner supports multiple user defined objectives. Result records have fields for `time`, `accuracy`, `energy`, `size`, `confidence`, and user defined data. The default objective is to minimize time. Many other objectives are supported, such as: maximize accuracy; threshold accuracy while minimizing time; and maximize accuracy then minimize size. The user can easily define their own objective by defining comparison operators and display methods on a subclass of Objective.

3.5 Search Driver and Measurement

OpenTuner is divided into two submodules, search and measurement. The search driver and measurement driver in each of these modules orchestrate most of the framework of the search process. These two modules communicate only through the results database. The measurement module is minimal by design and is primarily a wrapper around the user defined measurement function which creates results from configurations.

This division between search and measurement is motivated by a number of different factors:

- To allow parallelism between multiple search measurement processes, possibly across different machines. Parallelism is most important in the measurement processes since in most autotuning applications measurement costs dominate. To allow for parallelism the search driver will make multiple requests for desired results without waiting for each request to be fulfilled. If a specific technique is blocking waiting for results, other techniques in the ensemble will used to fill out requests to prevent idle time.
- The separation of the measurement modules is desirable to support online learning and sideline learning. In these setups, autotuning is not done before deployment of an application, but is done online as an application is running or during idle time. Since the measurement module is minimal by design, it can be replaced by an domain specific online learning module which periodically examines the database to decide which which configuration to use and records performance back to the database.
- Finally, in many embedded or mobile settings which require constrained measurement environments it is desirable to have a minimal measurement module which can easily be re-implemented in other languages without needing to modify the majority of the OpenTuner framework.

3.6 Results Database

The results database is a fully featured SQL database. All major database types are supported, and SQLite is used if the user has not configured a database type so that no setup is required. It allows different techniques to query and share results in a variety of ways and is useful for introspection about the performance of techniques across large numbers of runs of the autotuner.

4. Experimental Results

Project	Benchmark	Possible Configurations
GCC/G++ Flags	*all*	10^{806}
Halide	Blur	10^{52}
Halide	Wavelet	10^{44}
HPL	*n/a*	$10^{9.9}$
PetaBricks	Poisson	10^{3657}
PetaBricks	Sort	10^{90}
PetaBricks	Strassen	10^{188}
PetaBricks	TriSolve	10^{1559}
Stencil	*all*	$10^{6.5}$
Unitary	*n/a*	10^{21}

Figure 5. Search space sizes in number of possible configurations, as represented in OpenTuner.

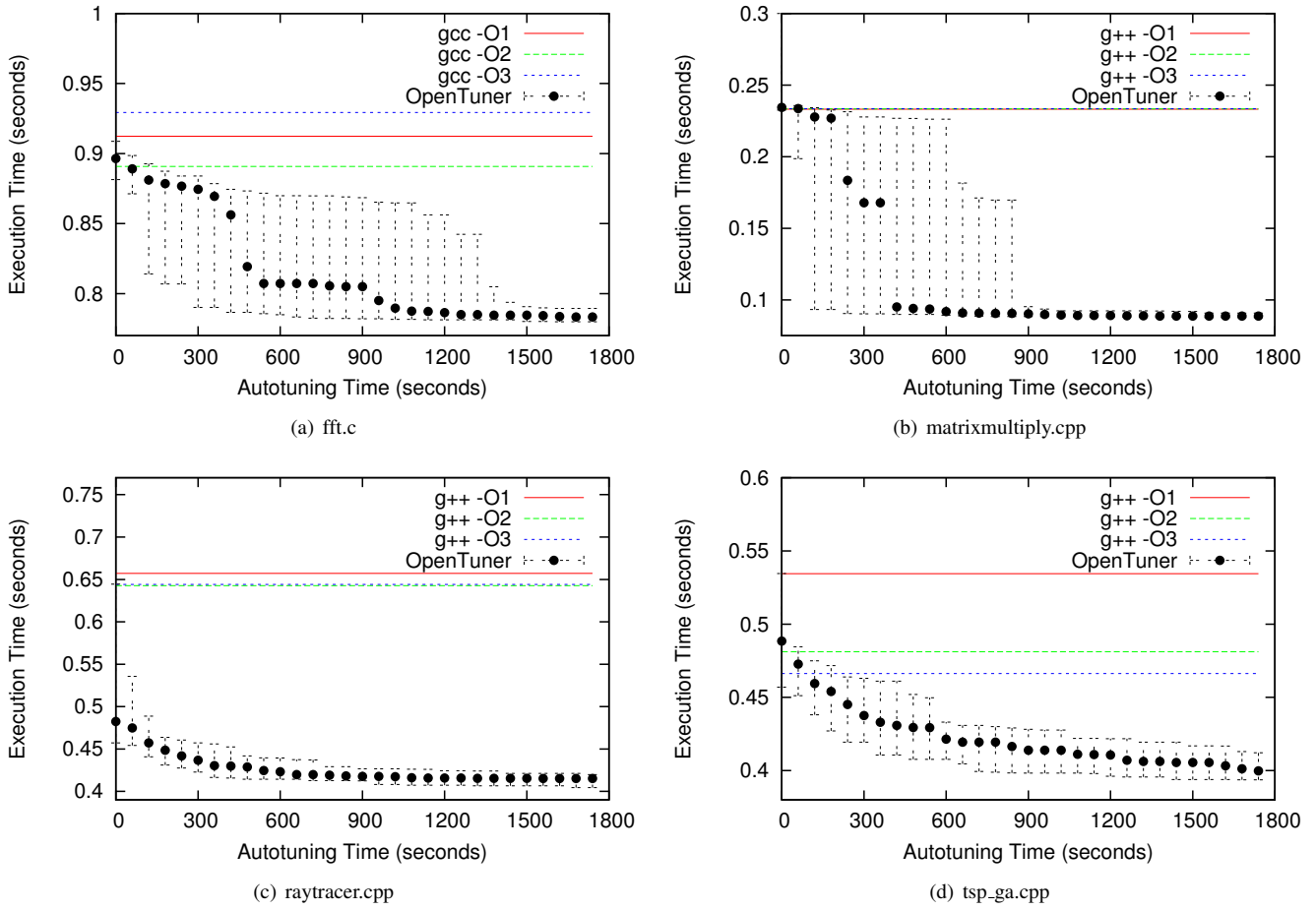
We validated OpenTuner by using it to implemented autotuners for six distinct projects. This section describes these six projects, the autotuners we implemented, and presents results comparing to prior practices in each project.

Figure 5 lists, for each benchmark, the number of distinct configurations that can be generated by OpenTuner. This measure is not perfect because some configurations may be semantically equivalent and the search space depends on the representation chosen in OpenTuner. It does, however, provide a sense of the relative size of each search space, which is useful as a first approximation of tuning difficulty.

4.1 GCC/G++ Flags

The GCC/G++ flags autotuner is described in detail in Section 3.1. There are a number features that were omitted from the earlier example code for simplicity, which are included in the full version of the autotuner.

First, we added error checking to gracefully handle the compiler or the output program hanging, crashing, running out of memory, or otherwise going wrong. Our tests uncovered a number of bugs in GCC which triggered internal compiler errors and we implemented code to detect, diagnose,



(a) fft.c (b) matrixmultiply.cpp

(c) raytracer.cpp (d) tsp_ga.cpp

Figure 6. GCC/G++ Flags: Execution time (lower is better) as a function of autotuning time. Aggregated performance of 30 runs of OpenTuner, error bars indicate median, 20th, and 80th percentiles. Note that in (b) the O1/O2/O3 and in (c) the O2/O3 lines are on top of each other and may be difficult to see.

and avoid error-causing sets of flags. We are submitting bug reports for these crashes to the GCC developers.

Second, instead of using a fixed list of flags and parameters (which the example does for simplicity), our full autotuner automatically extracts the supported flags from `g++ --help=optimizers`. Parameters and legal ranges are extracted automatically from `params.def` in the GCC source code.

Additionally, there were a number of smaller features such as: time limits to abort slow tests which will not be optimal; use of `LogInteger` parameter types for some values; a `save_final_config` method to output the final flags; and many command line options to autotuner behavior.

We ran experiments using `gcc 4.7.3-1ubuntu1`, on an 8 total core, 2-socket Xeon E5320. We allowed flags such a `-ffast-math` which can change rounding / NaN behavior of floating point numbers and have small impacts on program results. We still observe speedups with these flags removed.

For target programs to optimize we used: A fast Fourier transform in C, fft.c, taken from the SPLASH2 [34] benchmark suite; A C++ template matrix multiply, `matrixmultiply.cpp`, written by Xiang Fan [11] (version 3); A C++ ray tracer, `raytracer.cpp`, taken from the scratchpixel website [26]; and a genetic algorithm to solve the traveling salesman program in C++, `tsp_ga.cpp`, by Kristoffer Nordkvist [23], which we modified to run deterministically. These programs were chosen to span a range from highly optimized codes, like `fft.c` which contains cache aware tiling and threading, to less optimized codes, like `matrixmultiply.cpp` which contains only a transpose of one of the inputs.

Figure 6 shows the performance for autotuning GCC/G++ flags on four different sample programs. Final speedups ranged from $1.15\times$ for FFT to $2.82\times$ for matrix multiply. Examining the frequencies of different flags in the final configurations, we can see some patterns and some differences between the benchmarks. In all programs `-funsafe-math-optimizations` (and related flags) and `-O3` flags were very common. There were a number of flags that were only common only in specific benchmarks:

- `matrixmultiply.cpp`: `-fvariable-expansion-in-unroller` and `-ftree-vectorize`
- `raytracer.cpp`: `-fno-reg-struct-return`
- `fft.c`: `--param=allow-packed-store-data-races=1`, `-frerun-cse-after-loop`, and `-funroll-all-loops`
- `tsp_ga.cpp`: `--param=use-canonical-types=1` and `-fno-schedule-insns2`.

However these most common flags alone do not account for all of the speedup. Full command lines found contained typically 200 to 300 options and are difficult understand by hand.

4.2 Halide

Halide [28, 29] is a domain-specific language and compiler for image processing and computational photography, specifically targeted towards image processing *pipelines* that contain several stages. Halide separates the scheduling of the image processing stages from the expression of the kernels themselves, allowing expert programmers to dictate complex schedules that result in high performance.

The Halide project originally integrated an autotuner, which was removed from the project because it became too complex to maintain and was rarely used in practice. We hope that our new OpenTuner-based autotuner for Halide, presented here, will be easier to maintain, both because it benefits from some of the lessons learned from the original autotuner and because it provides a clear separation between the search techniques and the definition of the search space. Unfortunately, the original Halide autotuner cannot be used as a baseline to compare against, because the Halide code base has changed too much since its removal.

The autotuning problem in Halide is to synthesize execution schedules that control how Halide generates code. As an example, the hand-tuned schedule (against which we compare our autotuner) for the blur example is:

```
blur_y.split(y, y, yi, 8)
      .parallel(y)
      .vectorize(x, 8);
blur_x.store_at(blur_y, y)
      .compute_at(blur_y, yi)
      .vectorize(x, 8);
```

`blur_y(x, y)` and `blur_y(x, y)` are Halide functions in the program. The scheduling operators which the autotuner can use to synthesize schedules are:

- `split` introduces a new variable and loop nest by adding a layer of blocking. This operator makes the search space theoretically infinite; however, we limit the number of splits to at most 4 per dimension of each function, which is sufficient in practice. We represent each of these splits as a `PowerOfTwoParameter`, where setting the size of the split to 1 corresponds to not using the split operator.
- `parallel`, `vectorize`, and `unroll` cause the loop nest associated with a given variable in the function to be executed in parallel, SSE vectorized, or unrolled. OpenTuner represents these operators as an `EnumParameter` for each variable/function pair including temporary variables possibly introduced by splits to decide on an operator, including no operator as a choice.
- `reorder` / `reorder_storage` take a list of variables and reorganizes the loop nest order or storage order for those variables. We represent this is as a `Permutation-Parameter`, which includes all possible variables introduced by splits.
- `compute_at` / `store_at` cause the execution or storage for a given function to be embedded inside of the loop nest of a different function. We represent this as an `EnumParameter` with all legal function/variable pairs and special tokens for global and inline as options.

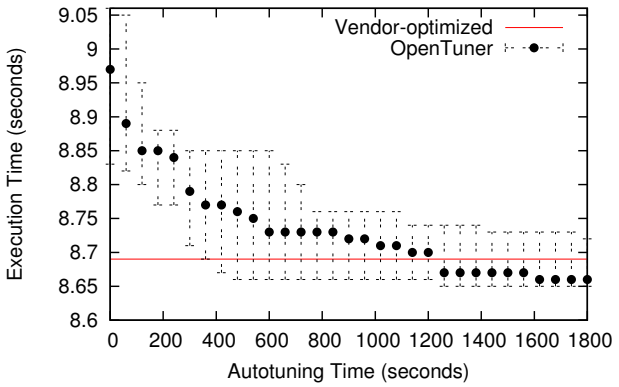


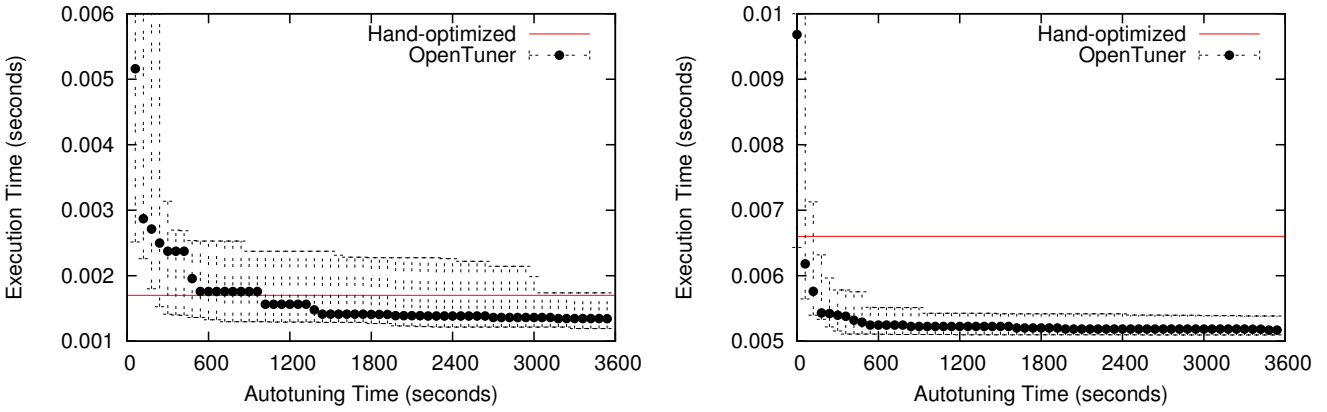
Figure 8. High Performance Linpack: Execution time (lower is better) as a function of autotuning time. Aggregated performance of 30 runs of OpenTuner, error bars indicate median, 20th, and 80th percentiles.

The most difficult parameter to search is `compute_at` because most choices combinations for this parameter will create invalid schedules are are rejected by the compiler. We created a custom domain specific technique which attempted to create more legal schedules by biasing the search of the parameter.

Figure 7 presents results for blur and the inverse Daubechies wavelet transform written in Halide. For both of these examples OpenTuner is able to create schedules that beat the hand optimized schedules shipping with the Halide source code. Results were collected on an 8-core Core i7 920 processor using a development build of Halide.

4.3 High Performance Linpack

The High Performance Linpack benchmark [10] is used to evaluate floating point performance of machines ranging from small multiprocessors to large-scale clusters, and is the evaluation criterion for the Top 500 [31] supercomputer benchmark. The benchmark measures the speed of solving a large random dense linear system of equations using distributed memory. Achieving optimal performance requires tuning about fifteen parameters, including matrix block sizes and algorithmic parameters. To assist in tuning, HPL in-



(a) Blur (b) Wavelet

Figure 7. Halide: Execution time (lower is better) as a function of autotuning time. Aggregated performance of 30 runs of OpenTuner, error bars indicate median, 20th, and 80th percentiles.

cludes a built in autotuner that uses exhaustive search over user-provided discrete values of the parameters.

We run HPL on a 2.93 GHz Intel Sandy Bridge quad-core machine running Linux kernel 3.2.0, compiled with GCC 4.5 and using the Intel Math Kernel Library (MKL) 11.0 for optimized math operations. For comparison purposes, we evaluate performance relative to Intel's optimized HPL implementation [1]. We encode the input tuning parameters for HPL as naïvely as possible, without using any machine-specific knowledge. For most parameters, we utilize `EnumParameter` or `SwitchParameter`, as they generally represent discrete choices in the algorithm used. The major parameter that controls performance is the blocksize of the matrix; this we represent as an `IntegerParameter` to give as much freedom as possible for the autotuner for searching. Another major parameter controls the distribution of the matrix onto the processors; we represent this by enumerating all 2D decompositions possible for the number of processors on the machine.

Figure 8 shows the results of 30 tuning runs using OpenTuner, compared with the vendor-provided performance. The median performance across runs, after 1200 seconds of autotuning, exceeds the performance of Intel's optimized parameters. Overall, OpenTuner obtains a best performance of 86.5% of theoretical peak performance on this machine, while exploring a miniscule amount of the overall search space. Furthermore, the blocksize chosen is not a power of two, and is generally a value unlikely to be guessed for use in hand-tuning.

4.4 PetaBricks

PetaBricks [3] is an implicitly parallel language and compiler which incorporates the concept of algorithmic choice into the language. The PetaBricks language provides a framework for the programmer to describe multiple ways of solving a problem while allowing the autotuner to determine which of those ways is best for the user's situation. The search space of PetaBricks programs is both over low level optimizations and over different algorithms. The autotuner is used to synthesize *poly-algorithms* which weave many individual algorithms together by switching dynamically between them at recursive call sites.

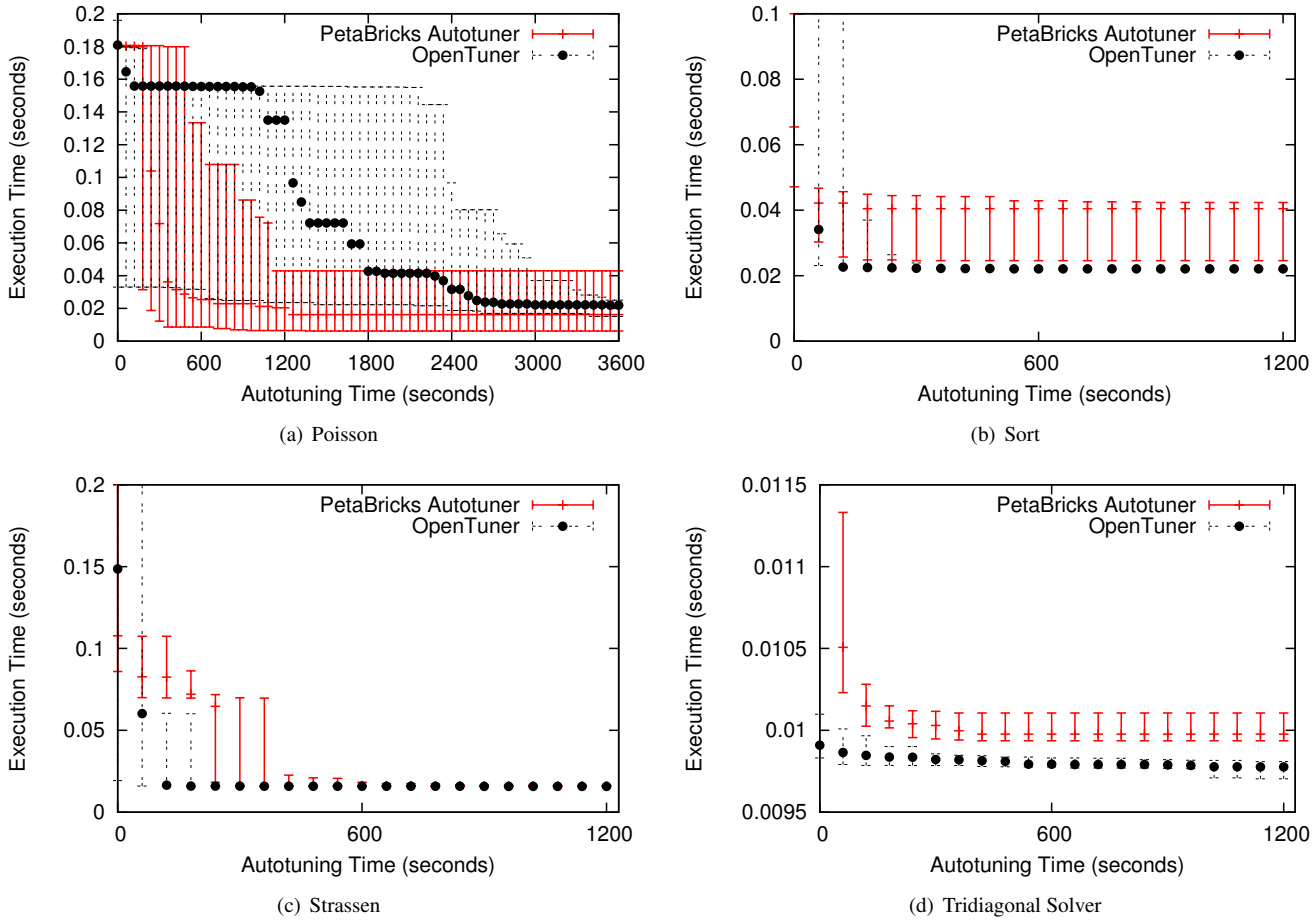
The primary components in the search space for PetaBricks programs are *algorithmic selectors* which are used to synthesize instances of poly-algorithms from algorithmic choices in the PetaBricks language. A selector is used to map input sizes to algorithmic choices, and is represented by a list of cutoffs and choices. As an example, the selector $[\text{InsertionSort}, 500, \text{QuickSort}, 1000, \text{MergeSort}]$ would correspond to synthesizing the function:

```
void Sort(List& list) {
  if(list.length < 500)
    InsertionSort(list);
  else if( list.length < 1000)
    QuickSort(list);
  else
    MergeSort(list);
}
```

where `QuickSort` and `MergeSort` recursively call `Sort` so the program dynamically switches between sorting methods as recursive calls are made on smaller and smaller lists. We used the general `SelectorParameter` to represent this choice type, which internally keeps track of the order of the algorithmic choices and the cutoffs. PetaBricks programs contain many algorithmic selectors and a large number of other parameter types, such as block sizes, thread counts, iteration counts, and program specific parameters.

Results using OpenTuner compared to the built-in PetaBricks autotuner are shown in Figure 9. The PetaBricks autotuner uses a different strategy that starts with tests on very small problem inputs and incrementally works up to full sized inputs [4]. In all cases, the autotuners arrive at similar solu-

[1] Available at `http://software.intel.com/en-us/articles/intel-math-kernel-library-linpack-download`.



(a) Poisson

(b) Sort

(c) Strassen

(d) Tridiagonal Solver

Figure 9. PetaBricks: Execution time (lower is better) as a function of autotuning time. Aggregated performance of 30 runs of OpenTuner, error bars indicate median, 20th, and 80th percentiles.

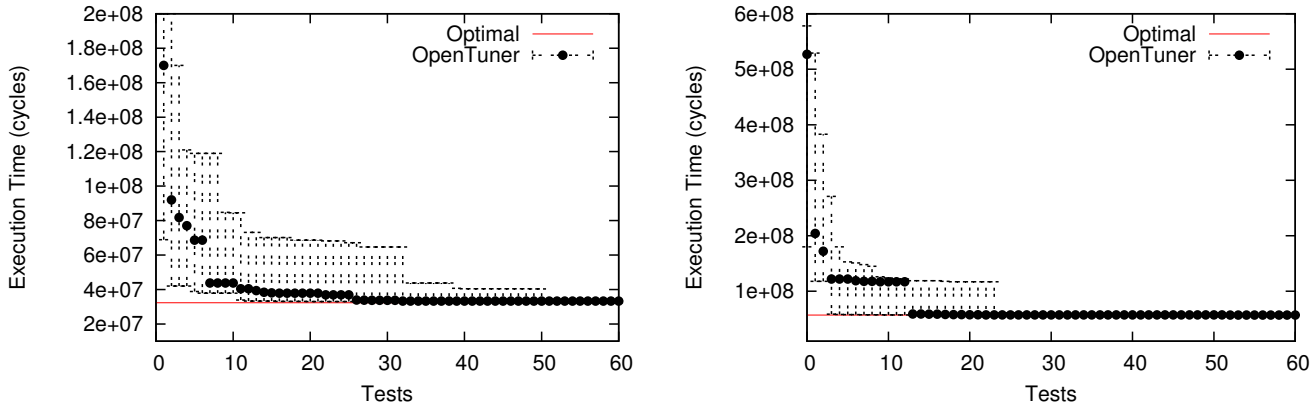
tions, and for Strassen, the exact same solution. For Sort and Tridiagonal Solver, OpenTuner beats the native PetaBricks autotuner, while for Poisson the PetaBricks autotuner arrives at a better solution, but has much higher variance.

The Poisson equation solver (Figure 9(a)) presents the most difficult search space. The search space for Poisson in PetaBricks is described in detail in [7]. It is a variable accuracy benchmark where the goal of the autotuner is to find a solution that provides 8-digits of accuracy while minimizing time. All points in Figure 9(a) satisfy the accuracy target, so we do not display accuracy. OpenTuner uses the *ThresholdAccuracyMinimizeTime* objective described in Section 3.4. The Poisson search space selects between direct solvers, iterative solvers, and multigrid solvers where the shape of the multigrid V-cycle/W-cycle is defined by the autotuner. The optimal solution is a poly-algorithm composed of multigrid W-cycles. However, it is difficult to obtain 8-digits of accuracy with randomly generated multigrid cycle shapes, but is easy with a direct solver (which solves the problem exactly). This creates a large "plateau" which is difficult for the autotuners to improve upon, and is shown near 0.16. The native PetaBricks autotuner is less affected by this plateau because it constructs algorithms incrementally bottom up; however the use of these smaller input sizes causes larger variance as mistakes early on get amplified.

4.5 Stencil

In [20], the authors describe a generalized system for autotuning memory-bound stencil computations on modern multicore machines and GPUs. By composing domain-specific transformations, the authors explore a large space of implementations for each kernel; the original autotuning methodology involves exhaustive search over thousands of implementations for each kernel.

We obtained the raw execution data, courtesy of the authors, and use OpenTuner instead of exhaustive search on the data from a Nehalem-class 2.66 GHz Intel Xeon machine, running Linux 2.6. We compare against the optimal performance obtained by the original autotuning system through exhaustive search. The search space for this problem involves searching for parameters for the parallel decomposition, cache and thread blocking, and loop unrolling for each kernel; to limit the impact of control flow and cache misalignment, these parameters depend on one another (for



(a) Laplacian (b) Divergence

Figure 10. Stencil: Execution time (lower is better) as a function of tests. Aggregated performance of 30 runs of OpenTuner, error bars indicate median, 20th, and 80th percentiles.

example, the loop unrolling will be a small integer divisor of the thread blocking). We encode these parameters as `PowerOfTwoParameters` but ensure that invalid combinations are discarded.

Figure 10 shows the results of using OpenTuner for the Laplacian and divergence kernel benchmarks, showing the median performance obtained over 30 trials as a function of the number of tests. OpenTuner is able to obtain peak performance on Laplacian after less than 35 tests of candidate implementations and 25 implementations for divergence; thus, using OpenTuner, less than 2% of the search space needs to be explored to reach optimal performance. These results show that even for problems where exhaustive search is tractable (though it may take days), OpenTuner can drastically improve convergence to the optimal performance with little programmer effort.

4.6 Unitary Matrices

As a somewhat different example, we use OpenTuner in an example from physics, namely the quantum control problem of synthesizing unitary matrices in $SU(2)$ in optimal time, using a finite control set composed of rotations around two non-parallel axes. (Such rotations generate the complete space $SU(2)$.)

Unlike other examples, which use OpenTuner as a traditional autotuner to optimize a program, the Unitary example uses OpenTuner to perform a search over the problem space as a subroutine at runtime in a program. The problem has a fixed set of operators (or controls), represented as matrices, and the goal is to find a sequence of operators that, when multiplied together, produce a given target matrix. The objective function is an accuracy value defined as a function of the distance of the product of the sequence to the goal (also called the *trace fidelity*).

Figure 11 shows the performance of the Unitary example on both easy and hard instances of the problem. For both types of problem instance OpenTuner is able to meet the accuracy target within the first few seconds. This is example shows that OpenTuner can be used for more types of search problems than just program autotuning.

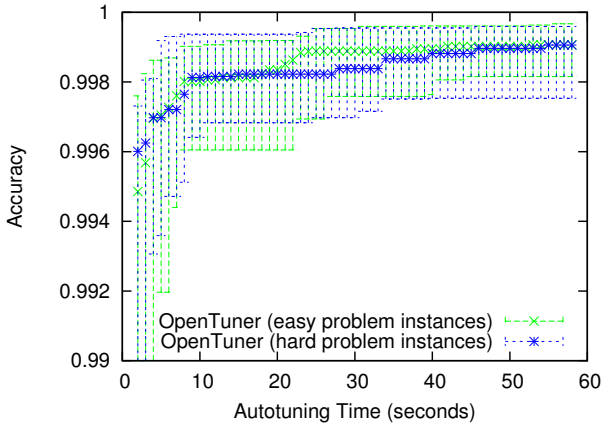


Figure 11. Unitary: Accuracy (higher is better) as a function of autotuning time. Aggregated performance of 30 runs of OpenTuner, error bars indicate median, 20th, and 80th percentiles.

5. Conclusions

We have shown OpenTuner, a new framework for building domain-specific multi-objective program autotuners. OpenTuner supports fully customizable configuration representations and an extensible technique representation to allow for domain-specific techniques. OpenTuner introduces the concept of ensembles of search techniques in autotuning, which allow many search techniques to work together to find an optimal solution and provides a more robust search than a single technique alone.

While implementing these six autotuners in OpenTuner, the biggest lesson we learned reinforced a core message of this paper of the need for domain-specific representations

and domain-specific search techniques in autotuning. As an example, the initial version of the PetaBricks autotuner we implemented just used a point in high dimensional space as the configuration representation. This generic mapping of the search space did not work at all. It produced final configurations an order of magnitude slower than the results presented from our autotuner that uses selector parameter types. Similarly, Halide's search space strongly motivates domain specific techniques that make large coordinated jumps, for example, swapping scheduling operators on x and y across all functions in the program. We were able to add domain-specific representations and techniques to OpenTuner at a fraction of the time and effort of building a fully custom system for that project. OpenTuner was able to seamlessly integrate the techniques with its ensemble approach.

OpenTuner is free and open source and as the community adds more techniques and representations to this flexible framework, there will be less of a need to create a new representation or techniques for each project and we hope that the system will work out-of-the-box for most creators of autotuners. OpenTuner pushes the state of the art forward in program autotuning in a way that can easily be adopted by other projects. We hope that OpenTuner will be an enabling technology that will allow the expanded use of program autotuning both to more domains and by expanding the role of program autotuning in existing domains.

Acknowledgments

We would like to thank Clarice Aiello for contributing the Unitary benchmark program. We gratefully acknowledge Jonathan Ragan-Kelley and Connelly Barnes for helpful discussions and bug fixes related to autotuning the Halide project.

This work is partially supported by DOE award DE-SC0005288 and DOD DARPA award HR0011-10-9-0009. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *International Symposium on Code Generation and Optimization*, pages 295–305, 2006.

[2] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Finding effective compilation sequences. In *LCTES'04*, pages 231–239, 2004.

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

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

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

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

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

[8] F. Chang and V. Karamcheti. A framework for automatic adaptation of tunable distributed applications. *Cluster Computing*, 4, March 2001. ISSN 1386-7857.

[9] M. Christen, O. Schenk, and H. Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *IPDPS*, pages 676–687. IEEE, 2011. ISBN 978-1-61284-372-8. URL `http://dblp.uni-trier.de/db/conf/ipps/ipdps2011.html#ChristenSB11`.

[10] J. J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK Benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003. ISSN 1532-0634. . URL `http://dx.doi.org/10.1002/cpe.728`.

[11] X. Fan. Optimize your code: Matrix multiplication. `https://tinyurl.com/kuvzbp9`, 2009.

[12] A. Fialho, L. Da Costa, M. Schoenauer, and M. Sebag. Analyzing bandit-based adaptive operator selection mechanisms. *Annals of Mathematics and Artificial Intelligence – Special Issue on Learning and Intelligent Optimization*, 2010. .

[13] A. Fialho, R. Ros, M. Schoenauer, and M. Sebag. Comparison-based adaptive strategy selection with bandits in differential evolution. In R. S. et al., editor, *PPSN'10: Proc. 11th International Conference on Parallel Problem Solving from Nature*, volume 6238 of *LNCS*, pages 194–203. Springer, September 2010. ISBN 978-3-642-15843-8. .

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

[15] G. Fursin, C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, B. Mendelson, E. Bonilla, J. Thomson, H. Leather, C. Williams, M. O'Boyle, P. Barnard, E. Ashton, E. Courtois, and F. Bodin. MILEPOST GCC: machine learning based research compiler. In *Proceedings of the GCC Developers' Summit*, Jul 2008.

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

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

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

[19] H. Jordan, P. Thoman, J. J. Durillo, S. Pellegrini, P. Gschwandtner, T. Fahringer, and H. Moritsch. A multi-objective auto-tuning framework for parallel codes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 10:1–10:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press. ISBN 978-1-4673-0804-5. URL `http://dl.acm.org/citation.cfm?id=2388996.2389010`.

[20] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12, 2010.

[21] S. A. Kamil. *Productive High Performance Parallel Programming with Auto-tuned Domain-Specific Embedded Languages*. PhD thesis, EECS Department, University of California, Berkeley, Jan 2013.

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

[23] K. Nordkvist. Solving TSP with a genetic algorithm in C++. `https://tinyurl.com/lq3uqlh`, 2012.

[24] M. Pacula, J. Ansel, S. Amarasinghe, and U.-M. O'Reilly. Hyperparameter tuning in bandit-based adaptive operator selection. In *European Conference on the Applications of Evolutionary Computation*, Malaga, Spain, Apr 2012.

[25] E. Park, L.-N. Pouche, J. Cavazos, A. Cohen, and P. Sadayappan. Predictive modeling in a polyhedral optimization space. In *IEEE/ACM International Symposium on Code Generation and Optimization*, pages 119 –129, April 2011.

[26] S. Pixel. 3D Basic Lessons: Writing a simple raytracer. `https://tinyurl.com/lp8ncnw`, 2012.

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

[28] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.*, 31(4):32:1–32:12, July 2012. ISSN 0730-0301.

[29] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6.

[30] C. Tapus, I.-H. Chung, and J. K. Hollingsworth. Active harmony: Towards automated performance tuning. In *In Proceedings from the Conference on High Performance Networking and Computing*, pages 1–11, 2003.

[31] Top500. Top 500 supercomputer sites. http://www.top500.org/, 2010.

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

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

[34] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *proceedings of 22nd Annual International Symposium on Computer Architecture News*, pages 24–36, June 1995.