Autotuning with Cloud Computing

Pedro Bruel — phrb@ime.usp.br
Alfredo Goldman — gold@ime.usp.br
Daniel Batista — batista@ime.usp.br

Instituto de Matemática e Estatística (IME)
Universidade de São Paulo (USP)
R. do Matão, 1010 – Vila Universitária, São Paulo – SP, 05508-090

Abstract

Test.

1 Introduction

2 Related Work

The Algorithm Selection Problem [15] consists in finding a *mapping* between *problems* and *algorithms* that minimizes the time to solve all instances in a problem set. The algorithms that compose a set can represent different abstractions, such as programs, heuristics, or configurations. The set of problems usually contains instances of a problem. Bougeret *et al.* [5] proved that the Algorithm Selection Problem is NP-complete when calculating static distributions of algorithms in parallel machines. Guo [8] proved the problem is undecidable in the general case.

Rice's conceptual framework formed the foundation of autotuners in various problem domains. In 1997, the PHiPAC system [3] used code generators and search scripts to automatically generate high performance code for matrix multiplication. Since then, systems tackled different domains with a diversity of strategies. Whaley *et al.* [17] introduced the ATLAS project, that optimizes dense matrix multiply routines. The OSKI [16] library provides automatically tuned kernels for sparse matrices. The FFTW [7] library provides tuned C subroutines for computing the Discrete Fourier Transform. In an effort to provide a common representation of multiple parallel programming models, the INSIEME compiler project [12] implements abstractions for OpenMP, MPI and OpenCL, and generates optimized parallel code for heterogeneous multi-core architectures.

Some autotuning systems provide generic tools that enable the implementation of autotuners in various domains. PetaBricks [1] is a language, compiler and autotuner that introduces abstractions, such as the "*either…or*" construct, that enable programmers to define multiple algorithms for the same problem. The ParamILS framework [11] applies stochastic local search methods for algorithm configuration and parameter tuning. The OpenTuner framework [2] provides ensembles of techniques that search spaces of program configurations. Bosboom *et al.* and Eliahu use OpenTuner to implement a domain specific language for data-flow programming [4] and a framework for recursive parallel algorithm optimization [6].

Gupta [9, 10].

2.1 OpenTuner

OpenTuner search spaces are defined by *Configurations*, composed of different *Parameter* types. Each type has restricted bounds, and implements its own manipulation functions, enabling the exploration of the search space. OpenTuner implements ensembles of optimization techniques that perform well in different problem domains.

Results found during the search process are shared between techniques through a common database. OpenTuner uses *meta-techniques* for coordinating the distribution of resources between techniques. An OpenTuner application can implement its own search techniques and meta-techniques.

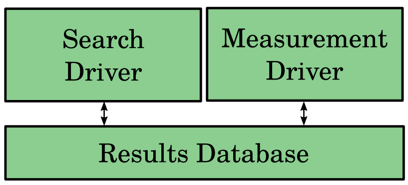


Figure 1: Simplified OpenTuner Architecture.

Figure 1 shows a high-level view OpenTuner's architecture. Measurement and searching are done in separate modules, called drivers, that share results through the database. The search driver requests measurements by registering configurations to the database. The measurement driver reads those configurations and writes back the desired results. Currently, the measurements are performed sequentially.

OpenTuner implements optimization techniques such as the Nelder-Mead [14] simplex method and Simulated Annealing [13]. OpenTuner implements a resource sharing mechanism that aims to take advantage of the strengths of each technique. A meta-technique must balance the exploitation of a technique that has produced good results in the past and the exploration of new, and possibly best, techniques.

3 Objectives

The objectives of this research project are the following:

- Implement an extension to the OpenTuner measurement driver, the *MeasurementClient*, that requests virtual machines in the Google Compute Engine to produce measurements of configurations;

- Implement a *MeasurementServer* that runs in the virtual machines. The server receives requests for results, computes them, and send them to the client;

- Normalize the results obtained from virtual machines to the local machine;

- Evaluate the performance of the implementation, using the benchmark applications provided with OpenTuner [2].

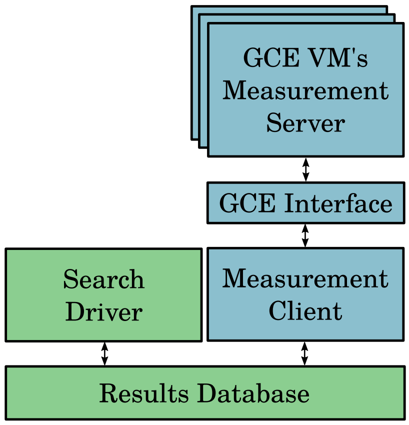
The remaining of this section describes the implementation objectives in detail, and reports the current state of the research. Section 4 describes the performance evaluations and the benchmark.

3.1 Measurement Server and Client

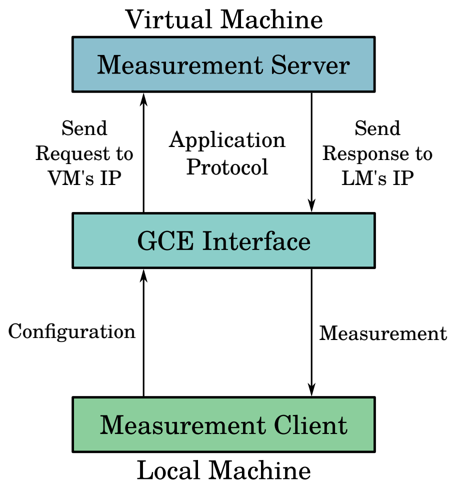
The interactions between the local and virtual machines will follow the client-server model. The local machine will run a measurement client, that will request results from various measurement servers running in virtual machines hosted at the Google Compute Engine.

Figure 2a shows the proposed architecture of an OpenTuner application running the measurement client and communicating with the measurement servers. Green boxes in the figure represent OpenTuner modules that will not be modified, and blue boxes represent new or modified modules.

Figure 2b shows, on a lower level of abstraction, the interactions between the measurement client and servers. A simple application protocol will be devised to mediate the requests and responses. The configurations will be sent to the server, that will run the correspondent program and respond with the measurement.



(a) A high-level view of the proposed architecture. Green boxes represent unmodified modules, blue boxes represent new or modified modules.



(b) A lower-level view of the proposed architecture, Illustrating the communication between server and client, mediated by the Google Compute Engine API.

Figure 2: Proposed OpenTuner Architecture, using the Google Compute Engine (GCE) API.

OpenTuner controls the execution flow of an application with the `main` function of the `TuningRunMain` class. This function calls the `main` function of the search driver, which runs the main loop of an OpenTuner application. The search driver generates configurations to be tested and saves them to the database. It them calls the `process_all` function of the measurement driver and blocks until the function returns.

The measurement driver is able to compile programs in parallel, but the measurements are done sequentially. Listing 1 shows the functions of the measurement driver that will have to be modified for the `MeasurementClient` to be able to process results with the Google Compute Engine.

During its initialization, the measurement client will initialize and configure the virtual machines, storing each measurement server's IP in the GCE interface. When the search driver requests for results, the `process_all` function will route the requests to the servers via the GCE interface. The interface will call the appropriate GCE Python API functions to send requests to the servers, and will wait for their responses.

The OpenTuner source code is available[1] under the MIT License.

```python
class MeasurementClient(MeasurementDriver):
    """
    Reads DesiredResults and requests VMs
    in the cloud to compute Results.
    """
    def __init__(self,
           measurement_interface,
           input_manager,
           **kwargs):
        super(MeasurementDriver, self).__init__(**kwargs)
        """
        Now, the client will instantiate and
        configure the VM Measurement Servers
        using the GCE interface.
        """

    def process_all(self):
        """
        Process all results in the
        database, by sending requests to
        VMs in the cloud and waiting
        for responses.
        """
```

Listing 1: Proposed modifications to the MeasurementDriver code.

[1]The code is hosted at GitHub: `github.com/jansel/opentuner`

3.2 Normalizing the Results

3.3 Using the Google Compute Engine

Simple experiments with GCE have been performed. A project was started and utility functions were implemented. The utilities include adding and removing virtual machines, configuring firewall access rules, configuring virtual machines with a startup script and obtaining a virtual machine's IP.

Each virtual machine is setup and starts running a simple Python TCP echo server at the `8080` port. Figure 2 shows the code for the server.

The utility functions and the measurement server's and client's code are available[2] under the GNU General Public License.

```python
#!/usr/bin/env python

import socket

TCP_IP = ''
TCP_PORT = 8080
BUFFER_SIZE = 1024

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((TCP_IP, TCP_PORT))
s.listen(1)

conn, addr = s.accept()

while 1:
    data = conn.recv(BUFFER_SIZE)
    if not data: break
    conn.send(data)
```

Listing 2: A simple echo server in Python.

[2] All code is hosted at GitHub:
`github.com/phrb/measurement-server`
`github.com/phrb/autotuning-gce`

4 Experiments

5 Research Schedule

6 Conclusion

References

[1] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. Petabricks: A language and compiler for algorithmic choice. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Dublin, Ireland, Jun 2009.

[2] 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 *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 303–316. ACM, 2014.

[3] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using phipac: A portable, high-performance, ansi c coding methodology. In *Proceedings of the 11th International Conference on Supercomputing*, ICS '97, pages 340–347, New York, NY, USA, 1997. ACM. ISBN 0-89791-902-5.

[4] Jeffrey Bosboom, Sumanaruban Rajadurai, Weng-Fai Wong, and Saman Amarasinghe. Streamjit: a commensal compiler for high-performance stream programming. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 177–195. ACM, 2014.

[5] Marin Bougeret, P-F Dutot, Alfredo Goldman, Yanik Ngoko, and Denis Trystram. Combining multiple heuristics on discrete resources. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.

[6] David Eliahu, Omer Spillinger, Armando Fox, and James Demmel. Frpa: A framework for recursive parallel algorithms. Master's thesis, EECS Department, University of California, Berkeley, May 2015.

[7] Matteo Frigo and Steven G Johnson. Fftw: An adaptive software architecture for the fft. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, volume 3, pages 1381–1384. IEEE, 1998.

[8] Haipeng Guo. *Algorithm selection for sorting and probabilistic inference: a machine learning-based approach*. PhD thesis, Citeseer, 2003.

[9] Abhishek Gupta, Laxmikant V Kalé, Dejan S Milojicic, Paolo Faraboschi, Richard Kaufmann, Verdi March, Filippo Gioachin, Chun Hui Suen, and Bu-Sung Lee. Exploring the performance and mapping of hpc applications to platforms in the cloud. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pages 121–122. ACM, 2012.

[10] Arpan Gupta, Paolo Faraboschi, Filippo Gioachin, Laxmikant V Kale, Richard Kaufmann, Bu-Sung Lee, Victor March, Dejan Milojicic, and Chun Hui Suen. Evaluating and improving the performance and scheduling of hpc applications in cloud. 2014.

[11] Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Thomas Stützle. Paramils: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36(1):267–306, 2009.

[12] 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.

[13] Scott Kirkpatrick, C Daniel Gelatt, Mario P Vecchi, et al. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[14] John A Nelder and Roger Mead. A simplex method for function minimization. *The computer journal*, 7(4):308–313, 1965.

[15] John R Rice. The algorithm selection problem. 1976.

[16] 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.

[17] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-89791-984-X.