Program Autotuning with Cloud Computing and OpenTuner

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. *The OpenTuner framework provides domain-agnostic tools for the implementation of autotuners. The optimization results are obtained sequentially by the measurement driver, that runs in a local machine. This paper presents an extension to the OpenTuner measurement driver, enabling it to leverage cloud computing resources from the Google Compute Engine. We compare the performance of our implementation using a diverse benchmark.*

1. Introduction

The program autotuning problem fits in the framework of the Algorithm Selection Problem, introduced by Rice in 1976 [1]. The objective of an autotuner is to select the best algorithm, or algorithm configuration, for each instance of a problem. Algorithms or configurations are selected according to performance metrics such as the time to solve the problem instance, the accuracy of the solution and the energy consumed. The set of all possible algorithms and configurations that solve a problem define a *search space*. Guided by the performance metrics, various optimization techniques search this space for the algorithm or configuration that best solves the problem.

Autotuners can specialize in domains such as matrix multiplication [2], dense [3] or sparse [4] matrix linear algebra, and parallel programming [5]. Other autotuning frameworks provide more general tools for the representation and search of program configurations, enabling the implementation of autotuners for different problem domains [6, 7].

The main contribution of this paper is the implementation of an extension to the OpenTuner framework [6] that enables it to leverage cloud computing resources. The interactions between the local and virtual machines follows the client-server model. The local machine runs a measurement client that requests results from various measurement servers running in virtual machines hosted at the Google Compute Engine. We compare the performance of our extension with the unmodified framework in a diverse benchmark of applications, identifying the problem domains that benefit from this cloud-based approach.

The rest of the paper is organized as follows. Section 2 discusses related work. Section **??** discusses the architecture of the OpenTuner framework. Section 4 presents the architecture of the measurement driver extension, the Google Compute Engine interface and the application protocol that mediates the interactions between *MeasurementClient* and *MeasurementServer*s. Section 5 discusses the result normalization strategies. Section 6 describes the experiments performed and the applications used in the benchmark. Section 7 presents a preliminary schedule. Section 8 concludes the proposal.

2. Related Work

Rice's conceptual framework [1] formed the foundation of autotuners in various problem domains. In 1997, the PHiPAC system [2] 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.* [3] introduced the ATLAS project, that optimizes dense matrix multiply routines. The OSKI [4] library provides automatically tuned kernels for sparse matrices. The FFTW [8] 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 [5] 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 [9] 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 [7] applies stochastic local search methods for algorithm configuration and parameter tuning. The OpenTuner framework [6] 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 [10] and a framework for recursive parallel algorithm optimization [11].

In a progression of papers [12, 13, 14], Gupta *et al.* provide experimental evaluations of the application of cloud computing to high performance computing, describing which kind of applications has the greatest potential to benefit from cloud computing. Their work highlights small and medium scale projects as the main beneficiaries of cloud computing resources.

3. OpenTuner

OpenTuner search spaces are defined by *Configurations*, that are composed of *Parameter* of various types. Each type has restricted bounds and manipulation functions that enable the exploration of the search space. OpenTuner implements ensembles of optimization techniques that perform well in different problem domains. The framework uses *meta-techniques* to coordinate the distribution of resources between techniques. Results found during search are shared through a database. An OpenTuner application can implement its own search techniques and meta-techniques, making the ensemble more robust. The source code is available[1] under the MIT License.

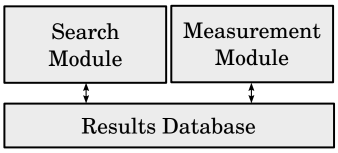

Figure 1. Simplified OpenTuner Architecture.

Figure 1 shows a high-level view of OpenTuner's architecture. Measurement and searching are done in separate modules, whose main classes are called *drivers*. 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 [15] simplex method and Simulated Annealing [16]. A resource sharing mechanism, called *meta-technique*, aims to take advantage of the strengths of each technique by balancing the exploitation of a technique that has produced good results in the past and the exploration of unused and possibly better ones.

4. 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.

[1] Hosted at GitHub: `github.com/jansel/opentuner`

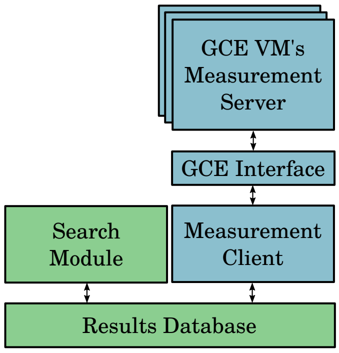

Figure 2. A high-level view of the proposed architecture. Green boxes represent unmodified modules, blue boxes represent new or modified modules.

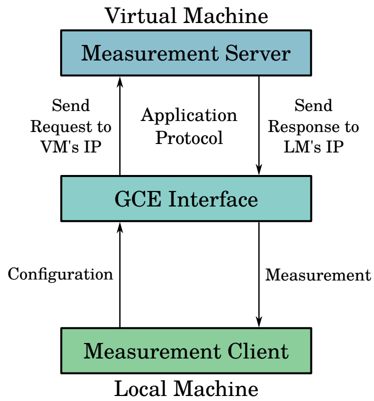

Figure 3. A lower-level view of the proposed architecture, Illustrating the communication between server and client, mediated by the Google Compute Engine API.

Figure 2 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 3 shows, on a lower level of abstraction, the interactions between the measurement client and servers. A Google Compute Engine interface and 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.

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

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

```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)
        """
        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 `MeasurementDriver`.

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 makes requests for results, the `process_all` function will route them to the servers via the GCE interface. The interface will call the appropriate GCE Python API functions, and wait for the responses.

Simple experiments with the Google Compute Engine have been performed. A project was started and utility functions were implemented, such as adding and removing virtual machines, configuring firewall access rules, configuring virtual machines with a startup script and obtaining a virtual machine's IP.

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

5. Normalizing the Results

The autotuner will optimize programs for a machine that will typically have an architectural specification different from the machines in the cloud. A normalization technique must be devised that enables the results found in the virtual machines to be valid for the local machine. Four preliminary approaches to this problem are discussed in the following.

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

The best approach will be experimentally determined, and could be combination of the approaches described here.

Use OpenTuner to Model Performance

Another autotuner could be implemented to optimize parameters of a simple performance model, that would associate a configuration's measurement and the virtual machine that produced it with a conversion function that transposes performance results to the target architecture.

Compose Ensembles of Virtual Machines

The cloud application could be composed of virtual machines with different architectures. The final performance measurement for a configuration would be built from some combination of the results obtained in these different virtual machines.

Simulate the Target Architecture

The target machine could be modeled by an architecture simulator such as *zsim* [17], a simulator for multi-core architectures available[3] under the GNU General Public License. Using a simulator would solve the normalization problem but introduce other problems, such as the simulator's accuracy and performance.

Run the Autotuner in the Cloud

Finally, the normalization problem could be sidestepped, at least in initial stages of research, by running the servers and clients in the cloud using the same kind of virtual machine.

6. Experiments

A benchmark of applications will be composed to compare the performance of the OpenTuner framework with and without the proposed modifications. An ideal benchmark would

[3]Hosted at GitHub: `https://github.com/s5z/zsim`

comprise the applications used to validate OpenTuner [6]. The normalization techniques devised for the results from virtual machines will be compared using the same benchmark.

We expect that the experiments provide insight into the situations when using cloud computing resources for autotuning is beneficial, and into the application and efficiency of the result normalization techniques.

7. Results

8. Conclusion

This proposal presented a research project aiming to extend the OpenTuner autotuning framework enabling it to leverage the cloud computing resources from Google Compute Engine. We believe that distributing measurements in the cloud will considerably speedup autotuning for every problem domain. We propose four approaches to solve the result normalization problem which would enable transposing the results obtained in virtual machines to a local machine. The benchmark for the experiments will be detailed during the development of the research. It will be composed preferably by problems that evidence strengths and weaknesses of the cloud environment.

References

[1] J. R. Rice, "The algorithm selection problem," 1976.

[2] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel, "Optimizing matrix multiply using phipac: A portable, high-performance, ansi c coding methodology," in *Proceedings of the 11th International Conference on Supercomputing*, ser. ICS '97. New York, NY, USA: ACM, 1997, pp. 340–347.

[3] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, ser. SC '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 1–27.

[4] R. Vuduc, J. W. Demmel, and K. A. Yelick, "Oski: A library of automatically tuned sparse matrix kernels," in *Journal of Physics: Conference Series*, vol. 16, no. 1. IOP Publishing, 2005, p. 521.

[5] 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 *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*. IEEE, 2012, pp. 1–12.

[6] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 2014, pp. 303–316.

[7] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle, "Paramils: an automatic algorithm configuration framework," *Journal of Artificial Intelligence Research*, vol. 36, no. 1, pp. 267–306, 2009.

[8] M. Frigo and S. 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*, vol. 3. IEEE, 1998, pp. 1381–1384.

[9] 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 *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Dublin, Ireland, Jun 2009.

[10] J. Bosboom, S. Rajadurai, W.-F. Wong, and S. 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*. ACM, 2014, pp. 177–195.

[11] D. Eliahu, O. Spillinger, A. Fox, and J. Demmel, "Frpa: A framework for recursive parallel algorithms," Master's thesis, EECS Department, University of California, Berkeley, May 2015.

[12] A. Gupta, L. V. Kalé, D. S. Milojicic, P. Faraboschi, R. Kaufmann, V. March, F. Gioachin, C. H. Suen, and B.-S. 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*. ACM, 2012, pp. 121–122.

[13] A. Gupta, P. Faraboschi, F. Gioachin, L. V. Kale, R. Kaufmann, B.-S. Lee, V. March, D. Milojicic, and C. H. Suen, "Evaluating and improving the performance and scheduling of hpc applications in cloud," 2014.

[14] A. Gupta, L. V. Kale, F. Gioachin, V. March, C. H. Suen, B.-S. Lee, P. Faraboschi, R. Kaufmann, and D. Milojicic, "The who, what, why, and how of high performance computing in the cloud," in *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, vol. 1. IEEE, 2013, pp. 306–314.

[15] J. A. Nelder and R. Mead, "A simplex method for function minimization," *The computer journal*, vol. 7, no. 4, pp. 308–313, 1965.

[16] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi *et al.*, "Optimization by simulated annealing," *science*, vol. 220, no. 4598, pp. 671–680, 1983.
[17] D. Sanchez and C. Kozyrakis, "Zsim: fast and accurate microarchitectural simulation of thousand-core systems," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013, pp. 475–486.