A New Program Autotuning Methodology Using Cloud Computing and OpenTuner

Pedro Bruel, Alfredo Goldman, Daniel Batista

[1]Instituto de Matemática e Estatística (IME)
Universidade de São Paulo (USP)
R. do Matão, 1010 Cidade Universitária, São Paulo SP, 05508-090

{phrb,gold,batista}@ime.usp.br

Abstract. *The OpenTuner framework provides domain-agnostic tools for the implementation of autotuners. It sequentially evaluates program configurations exploring search spaces that are commonly very large. This paper proposes a new methodology and a protocol for the distributed execution of autotuners. Both contributions are implemented in the OpenTuner framework as an extension to the OpenTuner measurement driver by distributing and parallelizing the measurement process via cloud computing resources. The performance evaluation shows that autotuners implemented with the extension are able to achieve good performance using few and cheap cloud computing resources, lowering the cost and the power consumption of producing an autotuned solution.*

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 defines a *search space*. Various optimization techniques search this space, guided by the performance metrics, for the algorithm or configuration that best solve the problem.

There are specialized autotuners for 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 OpenTuner framework [6] provides tools for the implementation of autotuners for various problem domains. It implements different search techniques that explore the same search space for program optimizations. Running and measuring program execution time — that is, the empirical exploration of the search space — is done sequentially. The framework provides support for parallel compilation.

The contributions of this paper are a new methodology and a protocol for the distributed execution of autotuners using cloud computing resources. The methodology and protocol are implemented as extensions to the OpenTuner framework distributing and parallelizing the exploration of optimization spaces by combining results obtained from virtual machines. A local machine (LM) runs the main OpenTuner application. Several virtual machines (VM) run measurement modules and provide results when requested. This distributed approach performs a more cost efficient exploration of the search space in comparison with the sequential execution.

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 a public cloud. The experiments in this paper used the Google Compute Engine (GCE). We compare the performance of our methodology, implemented as an OpenTuner extension, with the unmodified framework using two instances of the Travelling Salesperson Problem.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 describes the architecture of the OpenTuner framework. Section 4 presents the proposed methodology, the architecture of the measurement driver extension, the GCE interface and the application protocol. Section 5 discusses the result normalization strategies. Section 6 describes the experiments performed and the application used in the benchmark. Section 7 discusses the results obtained. Section 8 concludes the paper.

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.

To the best of our knowledge, the methodology presented in this paper is the first to propose using cloud computing resources to lower the autotuning costs using the OpenTuner framework.

3. OpenTuner

OpenTuner search spaces are defined by *Configuration*s, that are composed of *Parameter*s 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. OpenTuner's source code is available[1] under the MIT License.

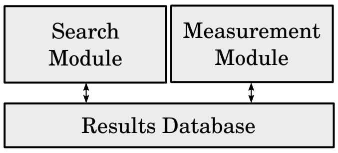

Figure 1. Simplified OpenTuner Architecture.

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. Using the measurement driver available in the framework, the measurements are performed sequentially.

Before running the user-defined measurement function the OpenTuner measurement driver calls compilation hooks that can be run in parallel using Python threads. When the autotuned program needs to be compiled only once, as in the Travelling Salesperson example shown in Section 6, the autotuner could respond to more simultaneous result requests if the measurement driver was able to also parallelize the runs of the user-defined measurement function.

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. Methodology and Protocol

The methodology and protocol proposed in this paper follow the client-server model, distributing measurements of program configurations between a group of virtual machines running *MeasurementServer*s in the cloud. The servers wait for measurement requests from a client, and maintain copies of the program to be autotuned and the user-defined function that measures configurations.

[1] Hosted at GitHub: `github.com/jansel/opentuner` [Accessed on 23 December 2015]

An interface encapsulates the communication from the client enabling a considerably lower implementation effort for the client. Figure 2 shows a rough estimate of the implementation effort for the three components needed to implement our methodology and protocol.

The machine running the OpenTuner autotuner runs a *MeasurementClient*, an extension of the native *MeasurementDriver*, that instead of compiling and running result requests locally, uses an interface to route requests to virtual machines and then saves the results to the local database.

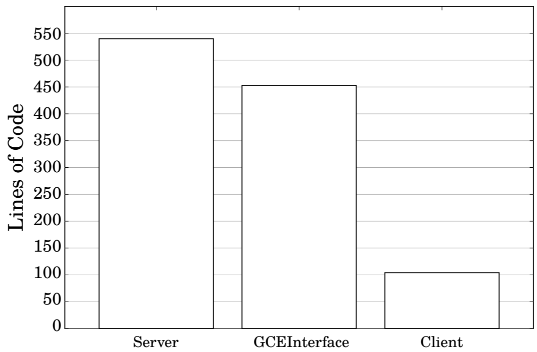

Figure 2. An estimate of the implementation effort, measured in lines of code.

Figures 3 and 4 present an overview of the architecture of the extension. Figure 3 presents the 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 4 shows, on a lower level of abstraction, the interactions between the measurement client and servers. The client requests results from the server through a wrapper of the GCE Python API. The GCE interface also encapsulates the application protocol used in the client-server communication.

The remaining of this section describes the extension implementation in further detail, the GCE interface and the application protocol.

4.1. Measurement Server and Client

OpenTuner controls the execution flow of an application with the `main` function of the *TuningRunMain* class. This function initialises 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 calls the `run_desired_results` function, which is able to run compilations in parallel but only sequential measurements. The modified *MeasurementDriver* initialises the GCE interface during its own initialisation. During execution the overridden `process_all` and `run_desired_results` functions route the result requests to the virtual machines using the *GCEInterface*.

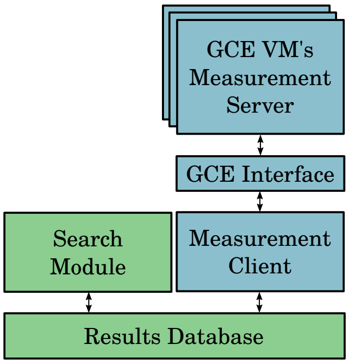

Figure 3. A high-level view of the architecture.

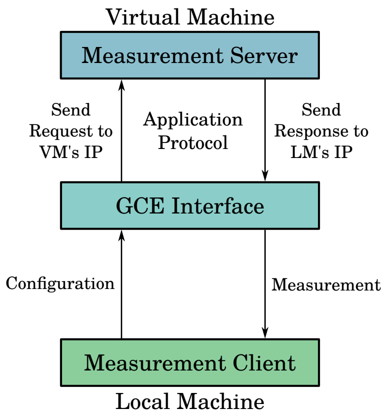

Figure 4. A lower-level view of the architecture.

An instance of the *MeasurementServer* runs in every virtual machine. The server waits for TCP connections from a single client.

4.2. GCE Interface

The interactions between the local *MeasurementClient* and the virtual machines' *MeasurementServers* are mediated by the *GCEInterface*, a wrapper of the GCE Python API. The interface starts and configures virtual machines storing each measurement server's IP.

The interface enables the *MeasurementClient* to request results from the servers without knowledge of the application protocol. Running our client-server methodology in another cloud environment would require a new interface that manages virtual machines in this environment, but no modifications are needed to the server or client.

4.3. Application Protocol

This section describes the text-based application protocol used in the client-server communications mediated by the *GCEInterface*. Note the `CLONE` message. The user's OpenTuner application must be a git project available via `HTTP`. The application will be cloned to the virtual machine by the server and used to obtain the autotuning results requested by the client.

Messages Table 1 shows all the messages in the protocol, a brief description of their meaning and their string format. The client must send a `MEASURE` message for each configuration that is measured. The server returns a unique ID that is used to retrieve the results when they are ready. This is done by sending a `GET` message.

Server Responses The server responds to each request with a message template and trailing, message-specific parameters. Responses always start with the correspondent command name and end with a newline character. Each response contains the current server status (`SERVER_STATUS`) and error code of the command (`ERROR_STATUS`). The optional argument list (`[ARGS..]`) contains the measurement result, for example, in the case of a successful `GET` response. Figure 5 shows the format of a server response.

Command	Function	Message
START	Sets the server's status to AVAILABLE	`'START'`
STOP	Sets the server's status to STOPPED	`'STOP'`
STATUS	Requests the server current status	`'STATUS'`
DISCONNECT	Disconnects from the server	`'DISCONNECT'`
SHUTDOWN	Disconnects and shuts the server down	`'SHUTDOWN'`
CLONE	Clones a git repository to the virtual machine	`'CLONE REPO_URL DIST_DIR'`
LOAD	Imports the user's MeasurementInterface into the server	`'LOAD TUNER_PATH INTERFACE_NAME'`
MEASURE	Computes the measurement for a given configuration	`'MEASURE CONFIG INPUT LIMIT'`
GET	Requests a configuration's result	`'GET RESULT_ID'`

Table 1. Server messages.

```
'COMMAND ERROR_STATUS SERVER_STATUS [ARGS..]  [MESSAGE]'
```

Figure 5. The format of a server response.

All the code implemented for the proposed methodology and protocol is publicly available at `github.com/phrb/measurement-server`, `github.com/phrb/gce_interface` and `github.com/phrb/measurement_client` under the GNU General Public License.

5. Result Normalization

The virtual machine instances available in public cloud computing services vary in multiple aspects, such as available memory, number of processing cores and disk size. When autotuning a program for a certain target machine, using instances hosted at a public cloud, it is typical that the target architecture will differ from the virtual machines'. Depending on the configuration of the cloud application the instances could also have different architectures.

To effectively leverage cloud computing resources for autotuning, a normalization technique must be devised that enables the results found in the virtual machines to be valid for the target machine. We present four approaches to this problem. The best approach for each problem domain must be experimentally determined, and could be a combination of the approaches described here.

Autotune Performance Models 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.

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.

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

Autotune 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

This section describes the Travelling Salesperson Problem (TSP) and the Google Compute Engine's virtual machines and project settings used to evaluate the efficacy of the proposed methodology and protocol. The performances of the autotuner for the TSP were measured in 8 different experimental settings. Each tuning run lasted 15 minutes, used 2 or 4 virtual machines, and was repeated 4 times. We also varied the number of result requests that each virtual machine in a tuning run processed, namely 1, 4, 8 or 16 requests per machine. The local machine used had an 8-core processor and 32GB of RAM.

6.1. Using the Google Compute Engine

All virtual machines used in the experiments had a single vCPU and 3.75GB of RAM (Google Compute Engine machine type `n1-standard-1`). All experiments were performed with machines from the `us-central1-f` zone. We built a virtual machine image with the latest stable Debian distribution and all dependencies installed, speeding up the virtual machines' initialization time.

6.2. Travelling Salesperson Problem

The instances of the TSP used in the experiments in this paper were obtained from TSPLIB [18]. A TSP solver was implemented as an OpenTuner application. The search space was defined by all the possible permutations, or tours, of cities where the first and last cities are the same. We used two instances, of size 532 and 85900.

7. Results

This section presents the results for all tuning runs for the TSP instances of sizes 532 and 85900. Figures 6 and 7 show the distributions of the costs of the best solutions found for the instance of size 85900 after 4 separate runs. It shows data for 2 and 4 virtual machines hosted in the cloud and receiving different numbers of simultaneous result requests. The

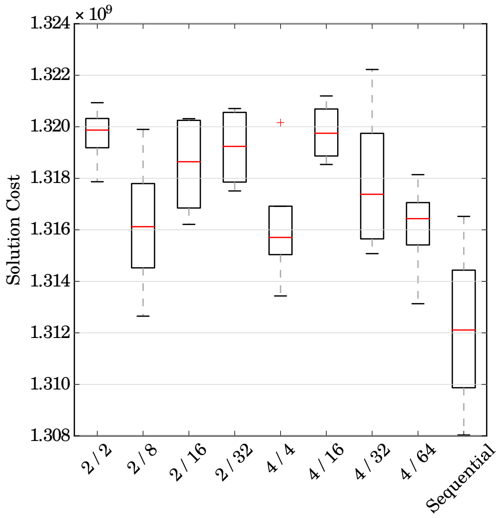

Figure 6. Boxplots of 4 runs solving an instance of size 85900 with different numbers of VMs and requests per VM.

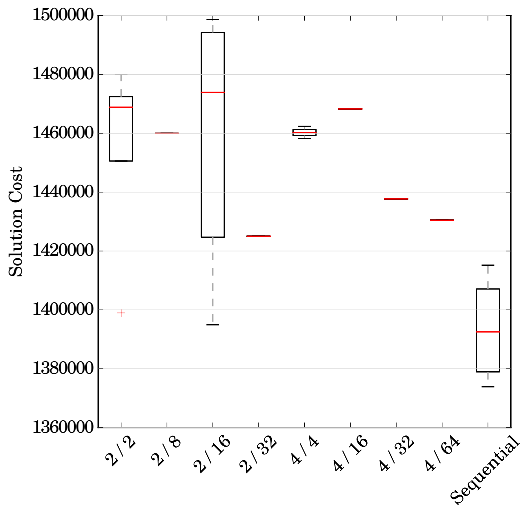

Figure 7. Boxplots of 4 runs solving an instance of size 532 with different numbers of VMs and requests per VM.

figure shows the distribution of the results obtained by running the autotuner in the target machine with the unmodified OpenTuner.

Figures 8 and 9 show the best runs in the instance of size 532 for the unmodified OpenTuner and for each virtual machine configuration. Figures 10 and 11 show the same measurements for the instance of size 85900. The time overhead in relation to the unmodified OpenTuner is clearly visible in Figures 8 to 11, and is due to the initialization of the cloud application.

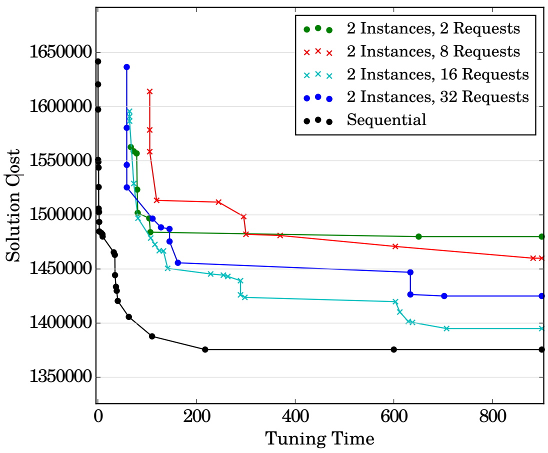

Figure 8. Measurements using two virtual machine instances, solving a TSP instance of size 532.

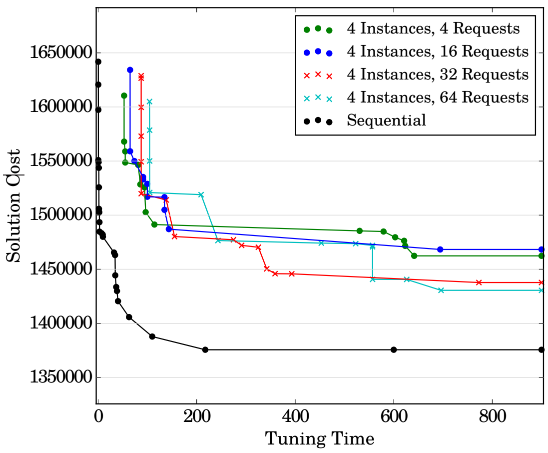

Figure 9. Measurements using four virtual machine instances, solving a TSP instance of size 532.

The tuning runs using our methodology and protocol depend on the initialization of a cloud application, and start to produce results later in the tuning run. The cost of running the same virtual machine instance used in our experiments with the Google Compute Engine were as low as US$0.038 [2] per hour, per instance. The cost of obtaining an autotuned solution of a given quality with our methodology is therefore considerably lower than the cost of acquiring a machine capable of finding the same solution.

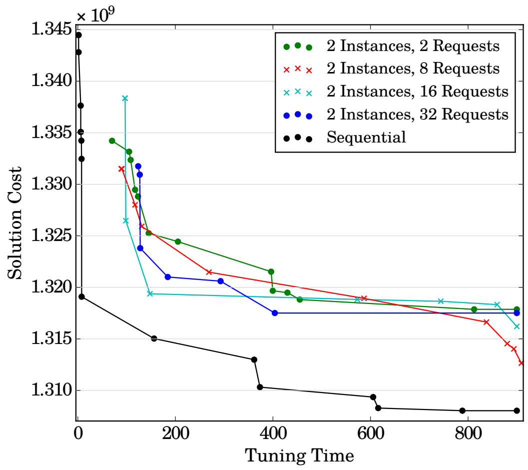

Figure 10. Measurements using two virtual machine instances, solving an instance of size 85900.

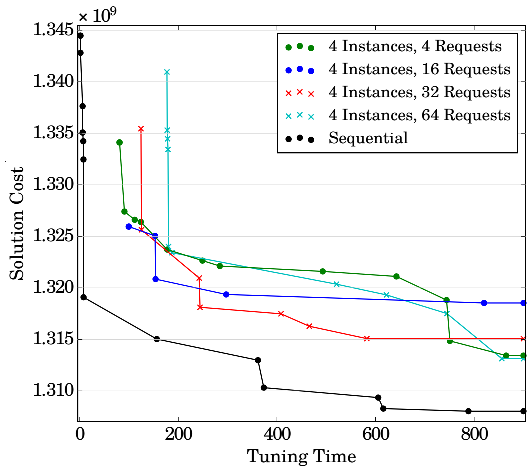

Figure 11. Measurements using four virtual machine instances, solving an instance of size 85900.

8. Conclusion

This paper presented a methodology and an application protocol for distributing autotuning measurements using cloud computing. We also present an implementation of the methodology and protocol, as an extension of the OpenTuner autotuning framework. We evaluate the performance and costs of our methodology and protocol using instances of the Travelling Salesperson Problem and the Google Compute Engine. Our results show that the methodology lowers the costs of finding a solution for the TSP. We propose four approaches to solve the result normalization problem which would enable transposing the results obtained in virtual machines to a local machine.

Future work will analyse the performance of our methodology and protocol in different problem domains, determining the ones that benefit the most from this approach. We will also study the normalization techniques we proposed.

To the best of our knowledge, the methodology and protocol presented in this paper are the first to propose using cloud computing resources to lower the autotuning costs using the OpenTuner framework.

References

[1] J. R. Rice, "The algorithm selection problem," in *Advances in Computers 15*, 1976, pp. 65–118.

[2] `cloud.google.com/compute/pricing#predefined_machine_types` [Accessed on 23 December 2015]

[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.

[18] G. Reinelt, "Tspliba traveling salesman problem library," *ORSA journal on computing*, vol. 3, no. 4, pp. 376–384, 1991.