ARPE: A tool to build equation models of computing systems

Martina Maggio
Lund University
Department of Automatic Control
martina.maggio@control.lth.se

Henry Hoffmann
University of Chicago
Department of Computer Science
hankhoffmann@cs.uchicago.edu

Abstract

An important challenge in the design and implementation of self-optimizing systems is that of finding a model that maps changes in a tunable parameter (or "knob") into an effect on the performance, power, or energy, of the overall system. This paper describes ARPE (Analyzing the Relationship between Parameters and Effectors), an open source tool to analyze the effect of parameter changes on the behavior of applications in a complex system with interrelated knobs.

We evaluate ARPE in several case studies on real systems with different sensors and parameters. Our results show that ARPE can help determine the best sensors for a system designed to predict application execution time. For space limitations, only one case study is here shown, demonstrating that the error of modeling energy consumption is limited to the range $0.1-10\%$ for previously unseen benchmarks.

1 Introduction

Autonomic computing has been proposed to relieve the burden on programmers, who are confronted by increasingly complicated systems. As described by Kephart and Chess, such "systems have hundreds of manually set nonlinear tuning parameters, and their number increases with each release" [13]. *Self-optimization* is usually the key to reduce user burden by automatically tuning such parameters to provide the best performance (or power, energy, etc).

At a high-level, the design of a model-based self-optimizing system typically follows three steps. First, experiments are run to collect data on the system's response to changing parameters. Second, the data produced by these experiments is analyzed to distill a model capturing the relationship between parameters and effects. Third, the model is used as a reference for the implementation of the self-optimizing system.

Unfortunately, this process can be very difficult for non-experts, who may recognize the benefits of building a self-optimizing system, but lack the expertise to do the analysis. Often, engineers base their model on assumptions (e.g., increasing cores will increase speed), but sometimes these assumptions do not hold (e.g., at some point increasing cores decreases speed due to communication overheads). Heo and Abdelzaher have recently demonstrated how assumptions which seem reasonable at design time may not hold at run time [8]. Given these issues, it can be difficult to develop the robust models needed to realize self-optimizing systems. Also, the process of gathering and analyzing data can be tedious. Thus, there is a need to automate the process of developing models for use in self-optimization modules.

This paper presents ARPE[1]: Analyzing the Relationship between Parameters and Effectors. ARPE users specify tunable parameters, valid ranges of those parameters, available sensors, and target applications. ARPE then performs data gathering and analysis on the target system. During the data gathering phase, ARPE runs a number of experiments with the system in different configurations. The data produced here is fed into ARPE's data analysis engine, which outputs a number of polynomial univariate and multivariate models describing the effects of tunable parameters on system behavior. We expect ARPE to be useful to non-experts who do not have the sophistication to develop robust models. We also expect ARPE to be helpful for experts in self-optimizing systems because it can reduce the tedium of collecting and analyzing data, leaving more time to focus on the mechanisms which make use of the models.

To demonstrate ARPE's use in practice, we present one case study. We test the generation of models which predict the amount of energy required to complete an application. ARPE obtains errors in the range of $0.1-10\%$ when the CPU consumption is used to model the ma-

[1] Available at `https://github.com/martinamaggio/arpe`.

chine energy consumption together with the application execution time. Overall, the results show that ARPE can be used to automate the process of creating accurate and robust models for the design and implementation of adaptive systems.

ARPE offers the following contributions:

- it automates the process of data collection in order to validate models for complex systems;
- it builds many models to fit the data obtained during, perform statistical analysis on the data and selects the best models to be used for the design of self-optimizing systems;
- it streamlines the process of designing large experimental campaigns to validate hypotheses on large systems.

The rest of this paper is organized as follows. Section 2 discusses the related literature. We present the analysis methodology in Section 3 and the experimental results in Section 4. In Section 5 we conclude the paper, highlighting future developments.

2 Related work

Self-optimizing systems have been proposed to solve many problems and their implementation has taken many forms; however, one commonality is that all such systems are built after first gathering and analyzing data. ARPE is motivated by the desire to create an open-source, extensible tool to handle these common tasks. This section presents several examples of data gathering and analysis in system design and then draws some conclusions that we incorporate into ARPE.

We draw on examples from hardware to software development. The evaluation of a new microprocessor design requires studying the impact of input data sets and workload composition [7]. One of the problems, in that domain, is finding a workload that represents the set of instructions that will be effectively executed on the microprocessor. Therefore, extensive benchmarking is undertaken where the parameters are different programs and input sets. Another example comes from High Performance Computing, where it is common to change an application parameter to adapt a running application. In [11] a threshold value is changed while executing parallel Monte Carlo ocean color simulations, while [6] presents a study on tuning Fast Fourier Transformations on graphic processing units. Also, Rahman et al. [15] and Tiwari et al. [18] studied the effect of compiler parameters on both performance and power/energy consumption for scientific computing. Program autotuning was studied in [12] where the focus is online parameter adaptation for tuning the application execution.

A lot of modeling and tuning effort has recently been devoted to the specific application of MapReduce [5]. Liu et al. propose compiler optimizations together with automatic parameter selection, recognizing the need for carefully selecting MapReduce parameters as there are about 150 tunable parameters [14]. Studies like this motivate ARPE development as, ideally the initial data gathering and analysis could be automated.

Further studies introduce cost-based optimization by expressing a set of choices containing, among other parameters, the number of map and reduce tasks, and the memory allocated to each of these [9]. The aim here is to find the best set of parameters to optimize MapReduce applications automatically [1]. Again, ARPE is motivated by the commonality of data collection and analysis, even when the results of that analysis are used to drive very different types of optimizations.

Another research direction in the specific area of MapReduce applications is aimed at modeling CPU consumption with multivariate linear regression [17] and computing the correlation between different applications [16]. Such studies are still preliminary, but many optimizations could be enabled if MapReduce applications have common CPU usage patterns; e.g., load placement for cloud computing infrastructures. However, further experimental evidence is needed with large benchmarking efforts, and ARPE is designed to help with exactly these types of efforts, while remaining independent from any individual project.

Clearly, self-optimizing systems have been designed and implemented in a number of forms, from hardware to software. While the particulars of different systems can vary, data gathering and analysis are common tasks. This development model leads to a situation where many different system developers recreate similar scripts for conducting experiments and processing the results. ARPE proposes separating the data gathering and analysis phases and creating a reusable, open-source tool that could handle these functions for different designers. Making the tool open-source and extensible allows different developers to contribute to it and helps the community learn from each other. While the importance of data gathering and analysis are well known, to the best of our knowledge, ARPE is the first proposal for an open source tool to aid the development of these crucial steps in the design of self-optimizing systems.

3 Methodology

This section describes the methodology that ARPE uses to produce and analyze data. The tool usage is divided into three different stages as sketched in Figure 1. In the first stage, the user inputs the parameters that are to be tested and the set of allowable values for each parameter, together with the benchmark to be executed. The outcome of this phase is the generation of an *execute.bash*

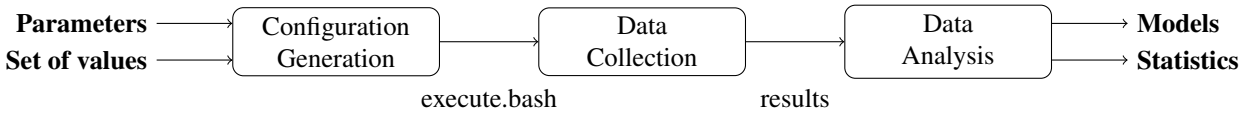

Figure 1: Typical usage of ARPE for collecting and analyzing data.

script that can be launched on the machine to start the data collection phase, aimed at obtaining measurements for the involved quantities with different set of parameters. Once the data collection is over, the results of the execution are written in the *results* directory and are ready to be analyzed. The user has two different options for data analysis. One is a Python model builder and the other is an Octave/Matlab model builder. Both options require specification of which parameters have to be used to build the model and the quantity (or quantities) that should be treated as output of the model. For example, a parameter might be number of cores assigned to an application, while the output quantity is total time to completion. The data analysis phase produces a number of possible polynomial uni- and multivariate models for the relationships between parameters and effectors. Users can compare these models and select one that best suits their needs.

3.1 ARPE configuration generation phase

The first phase in ARPE execution is configuration generation. Within configuration, the tool asks for a benchmark `command` to be executed for a certain number of `iterations`. The command can contain `parameters` to be changed when the execution has to be launched. For each parameter the user can specify different values or a range. For example, when testing the effect of the amount of cores assigned to a specific application, one can simply specify (a) a parameter `cores` standing for the number of cores and (b) a range for the parameter to be tested (for example 0 to 7). Each parameters configuration will be scheduled for execution a certain number of times, depending on the number of iterations chosen by the user.

3.2 ARPE data collection phase

The next phase in ARPE execution is data collection. In the generated script, there are three distinct phases for each test execution. In the "before" phase, some code is executed. This is, for example, where the hardware profiler is initialized. In the execution phase, the program is run in background with the current parameter set. In the "after" phase, some code to collect measurements is executed. The code to be executed for the run can contain the parameters and span over multiple lines. Both the instruction executed in the before and after phase can be changed. In fact, it is easy to extend ARPE to work with different sensors when these are available. For example, adding a power logger requires adding a single line starting the log at the end of the before phase and a line stopping the logger and saving the results at the beginning of the after phase.

Currently, CPU consumption is measured together with data retrieved by the hardware profiling with OProfile [4]. Among the low-level counters, the number of instructions retired will be used to demonstrate the tool capabilities. When a power measuring device is available, also power consumption is retrieved. All the recorded data are saved and used in the subsequent analysis phase. Intuitively, the number of combinations, given a set of parameters, grows exponentially with the amount of parameters to test and this phase can be very slow.

3.3 ARPE data analysis phase

ARPE performs two types of data analysis. It first builds univariate polynomial models for each parameter. It then builds multivariate models to capture interactions of parameters.

First, ARPE takes each parameter separately and finds the set of unique values it has during the experimental campaign. With these values, it tries to build a model that explains how the variation of that single parameter affects the measured quantity. For example, it may use the number of instructions retired to build a model to explain the duration of a specific computation. This process is repeated for every parameter with the same effector, producing different models that predict the variation of the measured quantity given the value of one specific parameter. In this initial step polynomial models are built.

For every model, ARPE fits the data with a polynomial model of the type

$$\psi_i(x_i, p) = x_0 + x_1 \cdot p + x_2 \cdot p^2 + \ldots + x_n \cdot p^n \quad (1)$$

where $\psi_i(x_i, p)$ is the predicted value for the measurable effect, p is the value of the parameter. The fit code finds the best values for the set of coefficients $x_i = [x_0, \ldots, x_n]$. ARPE tries to use models of different order n and selects the one which is the best fit for the data. It reports to the user the order of the selected model and some metrics about how good the fit was, i.e., the model accuracy.

In order to define the accuracy of a model, the tool

computes the mean error as

$$\frac{1}{m}\left[\sum_{i=1}^{m}\sqrt{(\psi_i(x_i,p)-y_i(p))^2}\right] \quad (2)$$

where x_i are the coefficients elaborated by the tool, p represents the value of the parameter, ψ_i is the predicted value and depends on the coefficients and the parameter, y_i is the measured value and m is the number of experiments. This metric will be used in the following to evaluate each type of model. Intuitively, the higher the error, the worse the model.

The second part of ARPE's data analysis consists of multivariate linear regression. The Ordinary Least Squares (OLS) algorithm is used to model the relationship between the entire vector of parameters and the completion time of the execution. Four different formulations are used. The forms of the models are:

$$\begin{array}{lrcl}
(a) & \psi_i(x_i,p_i) & = & x_0+x_1\cdot p_1+\ldots+x_n\cdot p_n \\
(b) & \psi_i(x_i,p_i) & = & x_0+x_1\cdot p_1+\ldots+x_n\cdot p_n+ \\
 & & & x_{n+1}p_1p_2+\ldots \\
(c) & \psi_i(x_i,p_i) & = & x_0+x_1\cdot p_1+\ldots+x_n\cdot p_n+ \\
 & & & x_{n+1}p_1p_2+\ldots+x_mp_1^2+\ldots \\
(d) & \psi_i(x_i,p_i) & = & x_0+x_1\cdot p_1+\ldots+x_n\cdot p_n+ \\
 & & & x_{n+1}p_1^2+\ldots
\end{array} \quad (3)$$

In the first formulation, (a), only linear terms are used. In (b), linear and interaction terms are considered. In (c) quadratic terms are added. Finally, (d) uses quadratic terms without the interaction terms. Finally, the tool computes some statistics with the generated models, showing the residuals (r), the mean square error (mse), and the R-square (rsquare) statistic for each model. The tool is designed so more statistics can be added very easily.

In the case of the execution time, for example, this multivariate regression is repeated three times with different measured values. The first set of models captures the relationship between the parameters and the wall-clock time from the run start to its end. The second set of models grasps the relationship between the involved parameters and the total run time summing up the time dedicated to each thread of the running application on every core. The third set, instead, models how the parameters influence the system time, which is the time spent in calling operating systems functions, for example for synchronization purposes. This analysis is performed in Matlab/Octave. ARPE also contains a python script that uses the statsmodels python library to provide multivariate linear regression model OLS and statistics for the data.

One can also imagine to have a more complex model, where a linear relationship can be captured between some values and execution time, where these values are different functions of the parameter set. It is quite easy to extend the modeling tool capabilities to include these models. For example, it is possible to have a single parameter which is the product of all the other parameters, or any combination. Example code is included in the ARPE distribution to demonstrate the extensibility of the model building module. We further explore this capability in the next section, to capture the relationship between CPU consumption, execution time and energy consumption.

4 Experimental results

Experimental setup. We test our approach on an Intel Xeon Processor E5530, equipped with a Wattsup device[2] installed allowing measurement of power and energy consumption.

Table 1: Summary of available profile counters.

`CPU_CLK_UNHALTED`	Clock cycles when not halted.
`INST_RETIRED`	Number of instructions retired.
`BR_INST_RETIRED`	Branch instructions retired.
`LLC_MISSES`	Last level cache misses.
`LLC_REFS`	Last level cache requests.

The `time` Unix command is used to measure execution time of the applications. The `pidstats` utility is used to gain insight on CPU consumption. OProfile is installed on both machines and Table 1 summarizes the events that can be profiled. In addition, when the code for the running applications is available, these programs can be instrumented to provide other data. For example, the PARSEC benchmarks [2] were instrumented with the Application Heartbeat framework [10] to provide high-level performance feedback.

Energy consumption. This section shows how ARPE can build a model of the energy consumption of a running application. We use three different applications to demonstrate the validity of the approach. To evidence the generality of ARPE we use previously unseen applications. We present detailed results for *sunflow* and *tradebeans* from the DaCapo benchmark suite [3]. We also use *lookbusy* to load the CPU. For each benchmark we show the averaged results over 25 runs.

Our aim is trying to find sensor measurements that we can use to predict the total energy consumed by the entire machine during the run of the application. This is important because the energy consumption is usually a key cost metric and it is valuable to build reliable models to predict it even before the application termination. The

[2] `http://www.wattsupmeters.com`

availability of such models would allow a self-optimizing system to limit energy consumption on the fly. Such a model can also be used in systems that provide guarantees about the total amount of energy consumed to run specific applications or whenever energy usage is to be optimized. Two very likely scenarios are energy consumption in data centers and optimized applications for mobile devices. As we will see, energy estimation differs from completion time estimation because it requires at least two sensors to build a reasonable prediction model.

We gather data from high-level hardware specific sensors, in the form of CPU consumption of the application. We also log power and energy information.

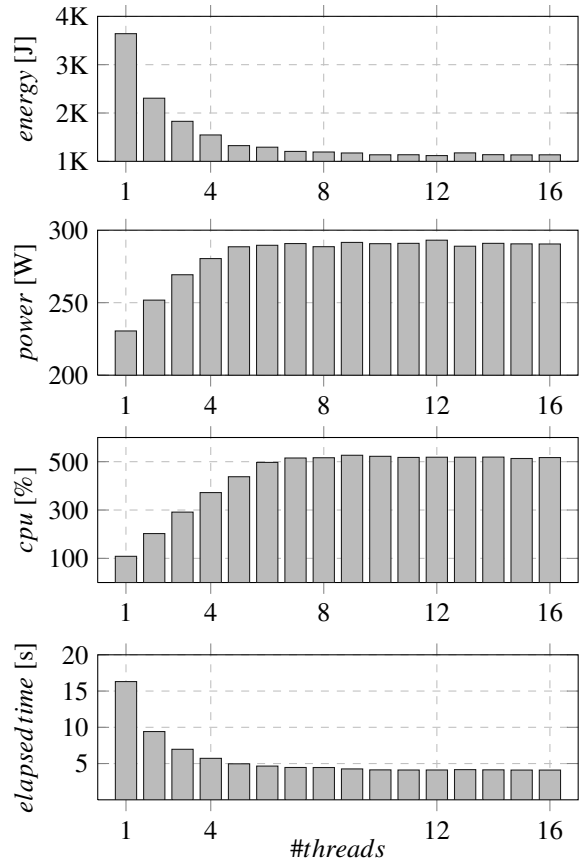

Figure 2: Results of the energy experiment with sunflow.

Figure 2 shows the result obtained varying the number of threads for the execution of the sunflow benchmark. As can be seen, the average power consumed during the benchmark execution depends linearly on the number of threads up to the maximum degree of parallelism for the application. Running with more than 6 threads does not provide any benefit. Neither the application execution time nor the amount of consumed energy decreases. However, up to the parallelism degree of the application, it is confirmed that the race to idle approach — allocating all the possible resources to compute and terminate faster — saves some energy. It is possible to automatically produce a very accurate model of the energy consumption based on the power consumption sensor and of the execution time. It is also possible to synthesize a model that is comparable in terms of accuracy to map the percentage of CPU consumed by the application and its execution time to the total energy consumed during the benchmark execution. In this case, the two parameters are combined into a single one (either $t_{exec} \cdot power$ or $t_{exec} \cdot \%cpu$) and the coefficient of this parameter is identified, together with an offset to take into account the idle power of the machine.

Table 2: Error (in joules) using different sensors, computed as specified in (2), lower is better.

	`power & time`	`CPU & time`
`sunflow`	16.10077	287.9064
`tradebeans`	59.9971	126.0322
`lookbusy`	20.6270	19.4846

The data for the benchmark is shown in Table 2. The model built with the CPU consumption produces an error which is about 10% of the maximum value. The model built using power consumption is obviously more accurate with an error of about 0.5% of the energy value. The analysis phase took 0.086 seconds as measured by Matlab command tic/toc. In this case, ARPE allows us to find a model that predicts the energy consumption with close to perfect accuracy given power information. Alternatively, ARPE also identifies a model that can predict energy *without* measuring power consumption on the fly. This second model gives up some accuracy, but does not require runtime power measurement.

The case of tradebeans is completely different. The application exposes a very limited amount of parallelism — as shown in Figure 3, in fact, the percentage of cpu consumed during its execution hardly reaches 125%. This means that in order to consume less energy, only two threads should be executed, not spawning useless threads onto the other cores. In this case, the analysis was able to detect an optimal point for the execution of the application. Also, the same model structure identified for sunflow explains very well the data for this application, both if the percentage of CPU is used and if the average power consumption is used.

The error data is reported Table 2. The model built with the CPU consumption produces an error about 1% of the maximum value. The model built using power consumption is more accurate with an average error of 0.5% of the energy value. The analysis phase took 0.002 seconds as measured by Matlab command tic/toc. When the same methodology is applied to other benchmarks from the DaCapo, the same two types of behavior are ex-

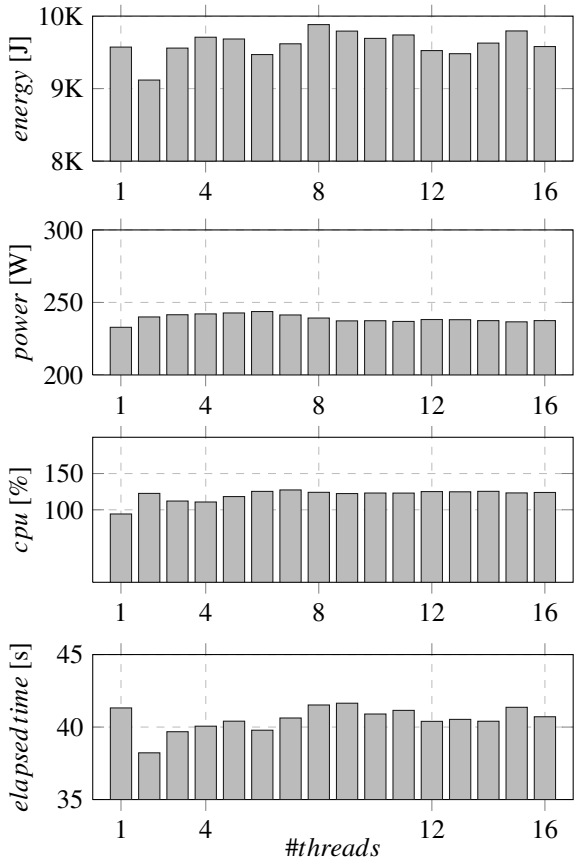

Figure 3: Energy results for tradebeans.

perienced. For example, h2 and xalan falls into the category where there is an optimal point in terms of energy consumption, while lusearch is similar to sunflow.

In order to avoid the degree of parallelism limitation, we conducted an experiment with *lookbusy*[3]. Lookbusy loads the CPU up to a certain parameter passed at the command line, which we chose to be 80%. Another parameter selects the number of CPUs to be loaded, and in our experiments we are vary that parameter from 1 to 8 and measuring the power and energy consumption. We also keep the execution time of the benchmark constant.

Figure 4 shows the result obtained varying the number of CPUs to be loaded with lookbusy. As can be seen, the amount of average power consumed during the benchmark execution depends linearly on the number of CPUs loaded. Table 2 summarizes the results of building a model from CPU consumption and power consumption in that case. The analysis phase took 0.002 seconds. In this case, using the two models results in almost identical errors, therefore they can be both considered reliable. The results confirm what previously seen with the execution of real applications. The error percentage in this case is even more limited, and it is around 0.1%

[3]Available at `http://www.devin.com/lookbusy/`.

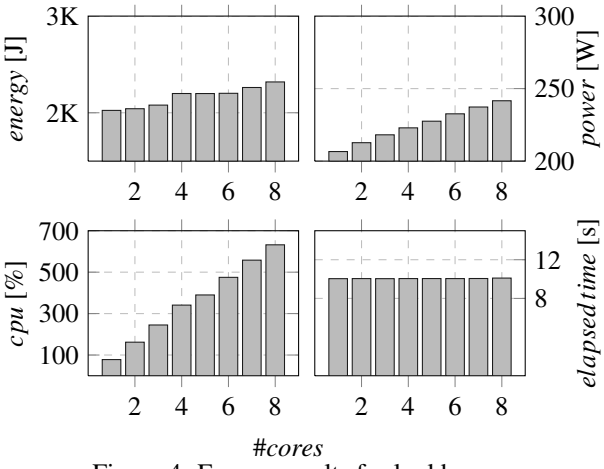

Figure 4: Energy results for lookbusy.

To summarize the contributions of this set of experiments, we used ARPE to find that using power consumption for energy prediction is ideal. However, if one does not have any power consumption sensor, it is possible to use the percentage of CPU consumed to have an estimate of the energy consumed by the benchmark. With the entire set of benchmarks we experienced errors in the range $0.1 - 10\%$.

5 Conclusion and future work

In this paper we presented ARPE, a tool for automating large experimental campaigns involving multiple parameters and to derive models from the obtained data. We developed the tool starting from our needs and we introduced different situations in which it proved useful. To prove the generality of the approach, we used well-known benchmarks coming from different suites, two diverse hardware configurations and a set of sensing and actuating capabilities.

More capabilities can be added for the data analysis phase, that is now limited to univariate and multivariate linear regression, for example non-linear models, identification techniques and neural networks. On the methodological side, the tool could be coupled with the capability of doing partial experiment modeling. In fact, it could be possible to add a flag to skip experiments that seems not to be needed based on previously collected results. This is matter for future extensions and has not yet been considered, although the design allows to plug in the logic easily.

Acknowledgement

This work was supported by the Swedish Research Council through the LCCC Linnaeus Center.

References

[1] BABU, S. Towards automatic optimization of mapreduce programs. In *Proceedings of the 1st ACM symposium on Cloud computing* (New York, NY, USA, 2010), SoCC '10, ACM, pp. 137–142.

[2] BIENIA, C., KUMAR, S., SINGH, J. P., AND LI, K. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (October 2008).

[3] BLACKBURN, S. M., GARNER, R., HOFFMANN, C., KHANG, A. M., MCKINLEY, K. S., BENTZUR, R., DIWAN, A., FEINBERG, D., FRAMPTON, D., GUYER, S. Z., HIRZEL, M., HOSKING, A., JUMP, M., LEE, H., MOSS, J. E. B., PHANSALKAR, A., STEFANOVIĆ, D., VANDRUNEN, T., VON DINCKLAGE, D., AND WIEDERMANN, B. The DaCapo benchmarks: java benchmarking development and analysis. *SIGPLAN Not. 41*, 10 (Oct. 2006), 169–190.

[4] COHEN, W. Tuning programs with oprofile. *Wide Open Magazine 1* (2004), 53–62.

[5] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. *Commun. ACM 51*, 1 (Jan. 2008), 107–113.

[6] DOTSENKO, Y., BAGHSORKHI, S. S., LLOYD, B., AND GOVINDARAJU, N. K. Auto-tuning of fast fourier transform on graphics processors. In *PPOPP* (2011), C. Cascaval and P.-C. Yew, Eds., ACM, pp. 257–266.

[7] EECKHOUT, L., VANDIERENDONCK, H., AND BOSSCHERE, K. D. Quantifying the impact of input data sets on program behavior and its applications. *J. Instruction-Level Parallelism 5* (2003).

[8] HEO, J., AND ABDELZAHER, T. Adaptguard: guarding adaptive systems from instability. In *Proceedings of the 6th international conference on Autonomic computing* (New York, NY, USA, 2009), ICAC '09, ACM, pp. 77–86.

[9] HERODOTOU, H., AND BABU, S. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *PVLDB 4*, 11 (2011), 1111–1122.

[10] HOFFMANN, H., EASTEP, J., SANTAMBROGIO, M. D., MILLER, J. E., AND AGARWAL, A. Application heartbeats for software performance and health. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2010), PPoPP '10, ACM, pp. 347–348.

[11] KAJIYAMA, T., DÁLIMONTE, D., AND CUNHA, J. C. Statistical performance tuning of parallel monte carlo ocean color simulations. In *The 13th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'12), Beijing, China* (12 2012), IEEE Computer Society, pp. 761–766. URL=http://www.pdcat2012.org/.

[12] KARCHER, T., AND PANKRATIUS, V. Run-time automatic performance tuning for multicore applications. In *Proceedings of the 17th international conference on Parallel processing - Volume Part I* (Berlin, Heidelberg, 2011), Euro-Par'11, Springer-Verlag, pp. 3–14.

[13] KEPHART, J. O., AND CHESS, D. M. The vision of autonomic computing. *Computer 36*, 1 (Jan. 2003), 41–50.

[14] LIU, J., RAVI, N., CHAKRADHAR, S. T., AND KANDEMIR, M. T. Panacea: towards holistic optimization of mapreduce applications. In *CGO* (2012), C. Eidt, A. M. Holler, U. Srinivasan, and S. P. Amarasinghe, Eds., ACM, pp. 33–43.

[15] RAHMAN, S. F., GUO, J., AND YI, Q. Automated empirical tuning of scientific codes for performance and power consumption. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers* (New York, NY, USA, 2011), HiPEAC '11, ACM, pp. 107–116.

[16] RIZVANDI, N., TAHERI, J., AND ZOMAYA, A. On using pattern matching algorithms in mapreduce applications. In *Parallel and Distributed Processing with Applications (ISPA), 2011 IEEE 9th International Symposium on* (may 2011), pp. 75 –80.

[17] RIZVANDI, N. B., TAHERI, J., MORAVEJI, R., AND ZOMAYA, A. Y. On modelling and prediction of total cpu usage for applications in mapreduce environments. In *ICA3PP (1)* (2012), Y. Xiang, I. Stojmenovic, B. O. Apduhan, G. Wang, K. Nakano, and A. Y. Zomaya, Eds., vol. 7439 of *Lecture Notes in Computer Science*, Springer, pp. 414–427.

[18] TIWARI, A., LAURENZANO, M. A., CARRINGTON, L., AND SNAVELY, A. Auto-tuning for energy usage in scientific applications. In *Proceedings of the 2011 international conference on Parallel Processing - Volume 2* (Berlin, Heidelberg, 2012), Euro-Par'11, Springer-Verlag, pp. 178–187.