# A+ Tuning: Architecture+Application Auto-tuning for In-Memory Data-Processing Frameworks

Han Wang[1], Setareh Rafatirad[2], and Houman Homayoun[1]
[1]University of California, Davis, CA, USA
[2] George Mason University, Fairfax, VA, USA
[1]{hjlwang,hhomayoun}@ucdavis.edu, [2]{srafatir}@gmu.edu

***Abstract*—Processing big data eventually leads to an upsurge in datacenters' power consumption, which is one of the pivotal concerns to be addressed. Many of the existing works focus on optimizing either power or performance, which is not the best parameter to consider for achieving high energy efficiency with low operational costs. Furthermore, the existing works require profiling of big data applications exhaustively and only consider tuning of either architectural or software parameters, often leading to sub-optimal settings. To cope up with the above-mentioned drawbacks of the existing works, we propose a system, A+ Tuning (Architecture + Application Auto-tuning) which enables to determine a close to optimal settings by simultaneously optimizing for Energy Delay product(EDP), representing energy efficiency. The proposed A+ Tuning involves a) profile the incoming unknown applications to different types (compute-bound, memory-bound and etc.) based on known applications classification result; b) co-locate the applications; and c) employs a machine learning based model to determine the optimal settings and tune from both architectural and application settings for the co-located applications. By applying the proposed A+ Tuning system, datacenters achieve up to 4× EDP improvement compared to fairshare methodology, and 2.5× compared with recent works such as BestConfig.**

***Keywords:* Auto-tuning, Machine Learning, Energy-efficiency.**

## I. Introduction

In the last two decades, the computing paradigms have experienced a tremendous change with vast amounts of data being amassed by various business units, and public organizations. Datacenters have emerged as a major facility to store and process such massive data and provide insights. To better utilize the datacenter's resources, several large-scale processing frameworks such as Hadoop [5], Spark [6], Tez [7], and Google Dataflow [8], have emerged. Among the aforementioned frameworks, Spark [6] is a prominent in-memory processing framework that has attracted attention from both academia and industry such as Facebook, Alibaba, and HortonWorks.

With the improvement in networking, storage, processing, and infrastructure management, scaling up datacenters is a preferable approach to cope with the big data challenge. However, adding more nodes to datacenters exacerbates power management and load-balancing issues. What's more, the increase of energy consumption and cooling costs give datacenters increasing pressure to achieve energy-efficiency. Since the hardware infrastructure cannot be scaled proportionately with the size of data, and the number of applications submitted to datacenters keeps growing, more applications need to be co-located at the node level. Therefore, the question of how to co-locate and configure data-intensive applications automatically for energy efficiency is becoming important. The existing works such as [1], [4] deploy tuning of software level parameters for higher performance, but are limited to tuning single application and cannot handle architectural parameter tuning. Furthermore, those methods involve time-consuming workload profiling leading to limited scalability.

To the best of our knowledge, no previous works comprehensively perform co-location and fine-tuning for unknown applications concerning the above challenges. In order to address aforementioned challenges, this work presents A+ Tuning system which can automatically co-locate and tune applications to improve energy efficiency. As shown in Table I, recent works in auto-tuning and co-locating big data applications are compared, showing the contribution and novelty of our work. Detailed contributions are listed below: (1) **comprehensive parameters** (both hardware architectural and application-level parameters are considered); (2) **ability to tune unknown applications** (profile applications firstly and classify them into different types); (3) **Machine-learning (ML) based self-tuning** (auto-tuning is achieved with the usage of ML techniques).

## II. Experimental Setup

### *A. Framework*

In this work, we use in-memory frameworks which are widely used for processing big data analysis, like Spark [6], Flink [9] and Tez [7]. Compared to Hadoop, Spark and Flink load data into memory and avoid frequent disk I/O to fasten data processing speed. Since Spark is more widely used, we conduct our experiment on these frameworks and show the detail of the proposed A+ Tuning system on Spark for a brief reason.

### *B. Workloads*

For evaluating the proposed A+ Tuning approach, we use the SparkBench [10] which includes a comprehensive set of applications such as machine learning, graph mining, data mining, data analysis, and pattern searching applications, which are frequently used in real-world applications. For each of the applications, three input data sizes are used: 1GB, 5GB and 10GB per node.

| Recent Work | Auto-tuning | Optimization Target | Profiling Method | Standalone Tuning | Co-located and Co-tuning | App Parameters | Arch Parameters |
|---|---|---|---|---|---|---|---|
| BestConfig [1] | ✓ | Performance | Multiple Times | ✓ |  | ✓ |  |
| LEO [2] | ✓ | Energy-efficiency | Multiple Times | ✓ |  |  | ✓ |
| Quasar [3] | ✓ | Performance | Partial |  | ✓ | ✓ |  |
| DAC [4] | ✓ | Performance | Multiple Times | ✓ |  | ✓ |  |
| **A+ Tuning** | **✓** | **Energy-efficiency** | **One Time** | **✓** | **✓** | **✓** | **✓** |

TABLE I: Relevant Works Comparison

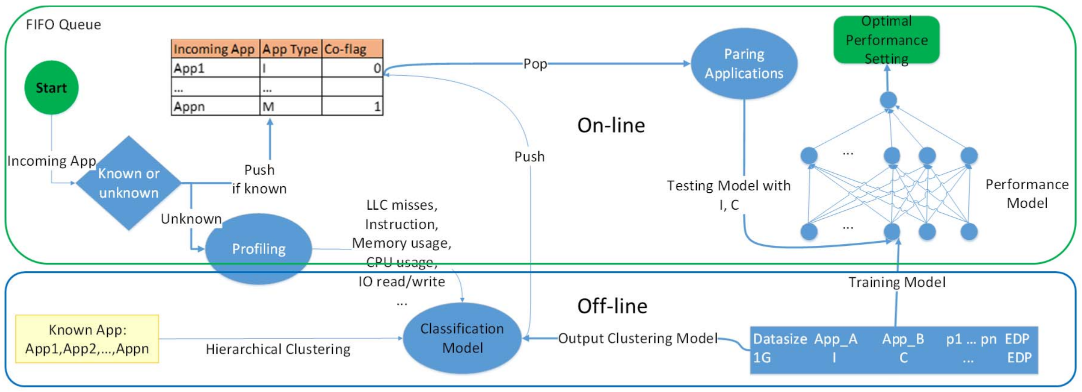

Fig. 1: The workflow of proposed A+ Tuning

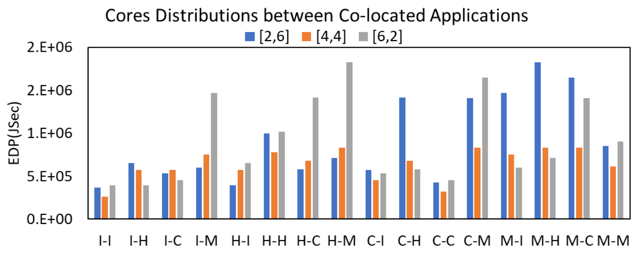

Fig. 2: Core distribution among different co-located applications A+ Tuning

## C. Hardware Platform and Software Settings

All experiments are conducted on a eight-node cluster which consists of eight Intel Xeon E5-2650 servers. Each Xeon server has eight processor cores and 8GB DRAM, three-level cache system. All of the experiments in this work are performed with a 10Gbit Ethernet network bandwidth, which is significantly higher than applications bandwidth requirements, thus the network does not become a bottleneck for the studied experiments. The operating system is Ubuntu 16.0.4 LST with Linux kernel 4.13, Hadoop version 2.7.4 and Spark version is 1.6.0. Details of the experiment environment are given in Table II. Table III lists tunable architectural and software level parameters for big data frameworks.

| | |
|---|---|
| Processor | Intel Xeon E5-2650, single socket-8 cores |
| Frequency | 1.2-2.6GHz |
| Storage | 480GB KINGSTON SHFS37A |
| Memory Capacity | 32GB DDR3 |
| Operating System | Ubuntu 16.0.4 LST |
| Hadoop version | 2.7.4 |
| Spark version | 1.6.0 |

TABLE II: Architectural node configurations

## D. Measurement tools

We use Perf [11] to measure the hardware (memory and processor) behavior. Perf is a profiling tool that can help to track the hardware performance counters. The performance counters data are collected for the entire run of each application. We collect OS-level performance information with DSTAT

| **Tuning Parameters** | **Range** | **Default Value** | Category |
|---|---|---|---|
| Frequency | 1.2-2.6GHz | 2.6GHz | Architecture |
| CPU cores | 1-8 | #core | Application |
| Memory Size | 1-8GB | 1GB | Architecture |
| Number of Partitions | 1-512 | 1 | Application |
| HDFS Block Size | 32-1024MB | 128MB | Application |

TABLE III: Application- and architecture-level tuning parameters

[12] tool. We use Wattsup PRO power meter to measure and record power consumption at one-second granularity. The power reading is for the entire system, including core, cache, main memory, hard disks, and on-chip communication buses. We collect the average power consumption for the studied applications and subtracted the system idle power to estimate the power consumption of the active state of cores.

## III. System design

In this section, we propose our methodology for energy efficiency by co-locating and auto-tuning big data applications, called A+ Tuning. The A+ Tuning has the following three modules: the application predicting module, the pairing module, and the self-tuning module. The application predicting module of A+ Tuning profiles an incoming application and collect run-time characteristics and classify the application to one of the application types. Then, the pairing module determines the applications that can be co-located. Lastly, the self-tuning module that comprises of an ML-based EDP model will give the optimal EDP settings to run the application without exhaustively searching the optimal one. More details are presented below.

| DataSize | App_A | AppName_B | p1 | ... | pn | EDP |
|---|---|---|---|---|---|---|
| 1GB | I | I | 1200MHz | ... | 64MB | $EDP_1$ |
| 1GB | I | C | 1800MHz | ... | 128MB | $EDP_2$ |
| 1GB | H | M | 2600MHz | ... | 256MB | $EDP_3$ |
| ... | ... | ... | ... | ... | ... | ... |

TABLE IV: Database structure

## A. Application Mapping

Firstly, known applications in datacenters are profiled and the architectural and system-level behaviors will be collected.

Then hierarchical clustering technique is employed to classify the known applications as compute-bound (C), memory-bound (M), I/O-bound (I) and hybrid referred to the combination of compute-bound and I/O bound (H). Secondly, the incoming unknown application are profiled and corresponding architectural and system-level behaviors are collected. Then the unknown application is classified by the clustering model built based on known applications as one of the types (C/M/I/H) based on known applications.

| | | |
|---|---|---|
| IPCs | Memory Footprint | Cycles |
| I/O read/write | LLC MPKI | CPU usage |

TABLE V: Features used to classify applications into C, H, M, I bound

**Algorithm 1:** Pairing System Pseudocode

```
Input: queue
Result: List of Apps to Co-locate
1  app ← Pop();
2  co-flag ←getFlag(app);
3  applist ← app;
4  tempstack ← NULL;
5  queuesize ←getSize(queue);
6  if queuesize == 0 then
7      return applist;
8  if co-flag != 0 then
9      return applist;
10 tempapp ←Pop(queue);
11 if getFlag(tempapp) ==1 then
12     applist ←applist +tempapp;
13 else
14     Push(tempstack,tempapp);
15 while getSize(tempstack) ≥0 do
16     tempstack ←Pop(tempstack);
17     Push(queue,tempstack);
18 return applist;
```

## B. Pairing Applications

Once each incoming unknown application has been profiled, the application will be paired for co-location. Since some applications may require to be executed solely by users, a co-location flag (co-flag) is used to indicate whether the application should be co-located or not. The co-flag can be either 0 or 1. When co-flag equals 0, incoming applications should not be co-located; when co-flag equals 1 which is the default value, incoming applications should be co-located.

In this work, incoming applications are kept in a FIFO queue that contains information about submitted job such as application name, data size and co-location flag. Algorithm 1 shows the pairing procedure. In this study, we consider the length of the queue described in line 6 to line 7, and then consider whether the application should be co-located or not (1 or 0), as shown in line 8 to line 9. For example, if the length of the queue is only 0, we decide to run the application individually. Else we use Pop() function to get one application. If the co-location flag equals 1, we add the application to applist, as in line 12; else we add it to tempstack. After getting enough applications to co-locate, we pop applications in tempstack and store them back to the queue again in line 16 and line 17 and return applications `applist` to co-locate. The last step returns the application pair (`applist`) and the other application with different co-location flag or co-location tag will be turned back to the queue. This procedure guarantees that applications never starve.

## C. Self-Tuning

After pairing applications, we get a pair like I-C, I-H, etc. Then we need to tune the paired applications according to pairing type. As shown in Figure 2, different pairs prefer various settings which cause very different EDP value. We propose a machine learning based approach (MLA) to predict the optimal EDP setting, which only requires to run **a part of** parameter settings and their results are employed as training dataset. For collecting the training dataset, each paired applications are executed under a part of settings and their EDP are collected as shown in Table IV. Then a Multilayer Perceptron (MLP), which is an artificial neural network model for creating a ML-based EDP model, is trained based on the training dataset. This approach significantly reduces the time required for finding the optimal setting, and also eliminates the need for executing applications under all possible settings. The inputs of MLP are the application's information which application type, like memory-bound, I/O-bound and so on. The output of MLP is an estimated EDP for the paired applications and the setting giving the optimal EDP by MLP model is the one A+ Tuning suggests.

## IV. Evaluation

In order to evaluate A+ Tuning system, we run unknown applications and then we use A+ Tuning to find the optimal configuration. We set co-location flag of incoming applications to 1, which represent applications should be co-located. We employ leave-one-out cross-validation (LOOCV) [13] technique for validating the mapping module. This means that we choose seven applications for training and test the model with the left one. Four applications are chosen as testing candidates. The four applications are: I (LinearR ), H (PCA), M (SVM), C (KMeans). In order to comprehensively evaluate the impact of data size on energy-efficiency tuning, we choose two different input data sizes (8GB and 80GB meaning 1GB and 10 GB per node). We use the following five baseline approaches for comparison with A+ Tuning:

- Co-located with randomly parameter setting [CL_Random]: in this baseline, we run co-located applications with one parameter setting chosen randomly from settings.
- Co-located with default setting [CL_Default]: running co-located applications with default settings.
- Co-located with fair-share [CL_Fairshare]: running co-located applications with fair-share of CPU processors and memory, i.e., four processors and 4GB memory for each application without frequency tuning.

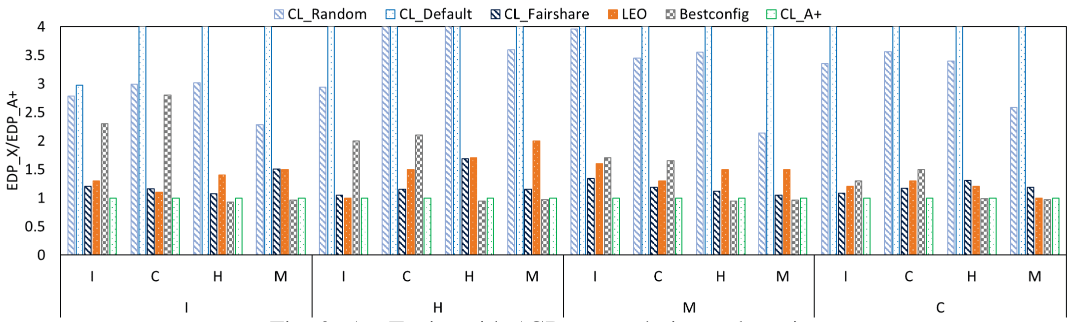

Fig. 3: A+ Tuningwith 1GB per node input data size

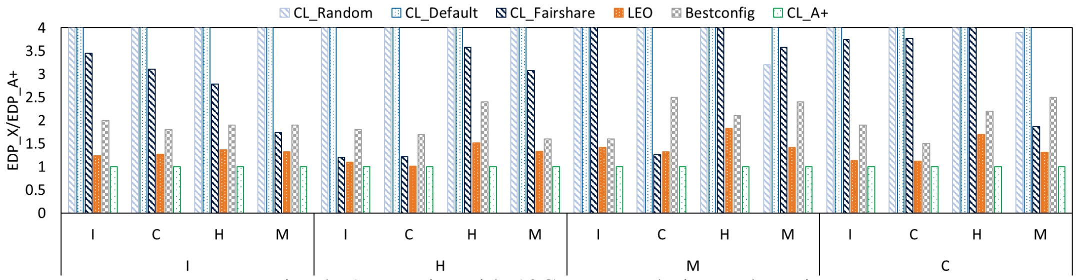

Fig. 4: A+ Tuningwith 10GB per node input data size

- Running applications with LEO presented in [2] [LEO]: running applications individually and adjusting resource setting according to continuously monitoring.
- Running applications with Bestconfig proposed in [1] [Bestconfig]: running applications individually with the optimal performance setting given by the proposed algorithm in former work [1].
- CL_A+: running co-located applications with parameter setting recommended by the proposed A+ tuning framework.

In Figure 3 and Figure 4, Y-axis represents the value of EDP_X (X represents CL_Random, CL_Default or CL_Fairshare, LEO and Bestconfig) normalized over EDP_A+, which is the EDP value of A+ tuning. The larger the ratio, the A+ Tuning performs better. The Figure shows that CL_Random and CL_Default have a much higher value than A+ tuning for both 8GB and 80GB input data size (1GB and 10 GB per node), meaning that A+ tuning performs better than randomly tuning or using default settings. When the input size is small (1GB per node), we observe that the EDP gap between CL_Fairshare and A+ tuning is around 20% on average. This is because applications with small data size have lower sensitivity to CPU and memory resources. However, when we increase the input data size to 80GB (10G per node), it can be observed that the EDP of CL_Fairshare is at most 4× than EDP of A+ Tuning. Comparing with Bestconfig [1] and LEO [2], A+ Tuning achieves up to 2.5 × and 2 × EDP respectively. The reason is that Bestconfig and LEO run applications individually and consider either application level or architectural level parameters only.

## V. Conclusion

This work presents the A+ Tuning to co-locate and fine-tune big data applications. By using the proposed A+ Tuning, datacenters only need to profile incoming unknown applications for once and achieve a close to optimal EDP, showcasing scalability for big data applications. Further to cope up with the large design space, the proposed A+ Tuning employs a ML-based technique to tune the co-located applications towards an optimal EDP. The results show that A+ Tuning is able to achieve up to 4× EDP improvement compared to fairshare for co-location, up to 2× EDP improvement compared to other race-to-idle method. Compared to recent tuning techniques, A+ Tuning shows up to 2.5× better energy efficiency.

## References

[1] Y. Zhu *et al.*, “Bestconfig: tapping the performance potential of systems via automatic configuration tuning,” in *Proceedings of the 2017 SoCC*.
[2] N. Mishra *et al.*, “A probabilistic graphical model-based approach for minimizing energy under performance constraints,” in *ACM SIGARCH Computer Architecture News*, vol. 43, pp. 267–281, ACM, 2015.
[3] C. Delimitrou *et al.*, “Quasar: resource-efficient and qos-aware cluster management,” *ACM SIGPLAN Notices*.
[4] Z. Yu *et al.*, “Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing,” in *he Twenty-Third ASPLOS*.
[5] T. White. O’Reilly Media Inc, 2012.
[6] M. Zaharia *et al.*, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in *9th USENIX NSDI*, 2012.
[7] B. Saha *et al.*, “Apache tez: A unifying framework for modeling and building data processing applications,” in *ACM SIGMOD International Conference on Management of Data*.
[8] T. Akidau *et al.*, “The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing,” *Proc. VLDB Endow.*
[9] P. Carbone *et al.*, “Apache flink: Stream and batch processing in a single engine,” *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.
[10] “Sparkbench,” in *https://github.com/CODAIT/spark-bench*.
[11] “Perf,” in *https://perf.wiki.kernel.org/index.php/Main Page*.
[12] “Dstat,” in *http://lintut.com/dstat-linux-monitoring-tools/*.
[13] P. Refaeilzadeh, L. Tang, and H. Liu, “Cross-validation,” in *Encyclopedia of database systems*, pp. 532–538, Springer, 2009.