Energy-Aware Node Selection for Cloud-Based Parallel Workloads with Machine Learning and Infrastructure as Code

Denis B. Citadin[1], Fábio Diniz Rossi[2], Marcelo C. Luizelli[3], Philippe O. A. Navaux[1] and Arthur F. Lorenzon[1]

[1]*Institute of Informatics, Federal University of Rio Grande do Sul, Brazil*
[2]*Campus Alegrete, Federal Institute Farroupilha, Brazil*
[3]*Campus Alegrete, Federal University of Pampa, Brazil*

fl

Keywords: Cloud Computing, Energy Efficiency, Infrastructure as Code, Artificial Intelligence.

Abstract: Cloud computing has become essential for executing high-performance computing (HPC) workloads due to its on-demand resource provisioning and customization advantages. However, energy efficiency challenges persist, as performance gains from thread-level parallelism (TLP) often come with increased energy consumption. To address the challenging task of optimizing the balance between performance and energy consumption, we propose *SmartNodeTuner*. It is a framework that leverages artificial intelligence and Infrastructure as Code (Iac) to optimize performance-energy trade-offs in cloud environments and provide seamless infrastructure management. SmartNodeTuner is split into two main modules: a *BuiltModel Engine* leveraging an artificial neural network (ANN) model trained to predict optimal TLP and node configurations; and *AutoDeploy Engine* using IaC with Terraform to automate the deployment and resource allocation, reducing manual efforts and ensuring efficient infrastructure management. Using ten well-known parallel workloads, we validate *SmartNodeTuner* on a private cloud cluster with diverse architectures. It achieves a 38.2% improvement in the Energy-Delay Product (EDP) compared to Kubernetes' default scheduler and consistently predicts near-optimal configurations. Our results also demonstrate significant energy savings with negligible performance degradation, highlighting *SmartNodeTuner* 's effectiveness in optimizing resource use in heterogeneous cloud environments.

1 INTRODUCTION

Cloud computing has been widely employed for executing parallel workloads across various domains, such as machine learning and linear algebra, due to its benefits of on-demand resource provisioning, customization, and resource control (Navaux et al., 2023). However, as these systems are usually heterogeneous and rely on energy-intensive data centers (Masanet et al., 2020), the challenge extends beyond performance optimization to include efficient resource utilization to reduce energy consumption and operating costs (Masanet et al., 2020). Given the characteristics of different applications, some workloads benefit more from running on nodes with fewer cores. In contrast, others require robust, high-core-count nodes for optimal performance and energy efficiency. For example, compute-bound applications with high scalability can fully utilize the resources of large-core nodes to maximize performance. On the other hand, memory-bound or less scalable applications often perform more efficiently on smaller nodes with lower core counts, as they minimize communication overhead and reduce contention for shared resources (Lorenzon and Beck Filho, 2019).

Moreover, the thread scalability of parallel workloads can also be constrained by their inherent characteristics. This means that running the workloads with the maximum number of cores available in the node will not always deliver the best performance and energy efficiency outcomes (Suleman et al., 2008). Workloads with limited thread-level parallelism (TLP), such as those with high inter-thread communication or synchronization requirements, may not achieve significant performance gains even on nodes with a high number of cores. In these cases, increasing the number of threads can lead to diminishing returns, where the overhead of synchronization and resource contention offsets the benefits of parallel execution. Consequently, determin-

Citadin, D. B., Rossi, F. D., Luizelli, M. C., Navaux, P. O. A. and Lorenzon, A. F.
Energy-Aware Node Selection for Cloud-Based Parallel Workloads with Machine Learning and Infrastructure as Code.
DOI: 10.5220/0013418500003950
In *Proceedings of the 15th International Conference on Cloud Computing and Services Science (CLOSER 2025)*, pages 49-60
ISBN: 978-989-758-747-4; ISSN: 2184-5042

ing the ideal TLP degree and selecting the appropriate node for execution is essential to effectively balancing performance and energy efficiency in heterogeneous cloud environments.

To alleviate the burden on software developers and end-users in defining these execution parameters, Infrastructure as Code (IaC) offers an effective solution. IaC enables resource provisioning and configuration automation by expressing infrastructure requirements as code. This approach allows developers to define the desired infrastructure state —- such as node selection and thread allocation—in a declarative manner, leaving the deployment and setup to IaC tools. For instance, workloads that scale effectively with higher thread counts can be automatically deployed on nodes with a large number of cores using IaC scripts. Conversely, workloads with limited scalability can be allocated to nodes with fewer cores, optimizing both performance and energy consumption.

Given the complexities in identifying the best combinations of computing nodes and TLP for executing parallel workloads in cloud heterogeneous environments, this paper makes three key contributions. (*i*) a *BuildModel engine* that employs an artificial neural network (ANN) based model to determine the combination that delivers the best balance between performance and energy consumption for each parallel workload. The ANN model is trained on a dataset containing representative hardware and software metrics from workloads with distinct computational and memory access characteristics. (*ii*) *AutoDeploy engine*, which relies on IaC to automate the deployment of workloads using the combinations predicted by the predictor engine across varied cloud resources. For that, we rely on the Terraform tool to simplify node configuration and management, reducing the need for manual intervention while providing control over how resources are distributed. And, (*iii*) *SmartNodeTuner*, a framework that integrates both engines to automate the selection of the most suitable computing node and TLP degree for running parallel workloads on cloud environments.

To validate *SmartNodeTuner*, we conducted experiments with ten well-established applications spanning various domains, all deployed on a private cluster featuring nodes with different architectural characteristics. Throughout our validation, *SmartNodeTuner* improved the Energy-Delay Product (EDP) by 38.2% compared to *Kube-Scheduler*, the default scheduler in Kubernetes. Additionally, compared with an exhaustive search method that considers every possible node and thread configuration for each application, *SmartNodeTuner* predicted configurations that fell within the top two optimal solutions in over 80% of instances. Furthermore, we also show that *SmartNodeTuner* provides significant energy savings while having minimal impact on the workloads' performance.

2 BACKGROUND

2.1 Cloud Computing

Cloud computing has become the standard for application deployment due to its on-demand resource availability over the Internet (Liu et al., 2012). While resource provisioning appears seamless to users, various technologies work in the background to ensure essential features like elasticity and high availability (Márquez et al., 2018). Initially, cloud systems struggled to meet the demands of compute-intensive applications needing rapid response times, such as Big Data and Analytics, due to virtualization overhead (Barham et al., 2003). This led to the adoption of lightweight container technologies like Docker, which closely approaches the performance of non-virtualized systems. Docker has become the preferred platform for developing, packaging, and running containerized applications, encapsulating all necessary components like libraries and binaries for streamlined execution. Docker is particularly useful for deploying parallel applications, as it creates isolated environments with all required dependencies while minimizing the overhead typical of traditional virtualization. This efficiency makes Docker ideal for resource-demanding HPC applications, enabling parallel applications to scale effectively across multiple nodes and maximizing cloud-based HPC infrastructure use.

2.2 Infrastructure as Code - IaC

Infrastructure as Code (IaC) is an approach that enables software developers and administrators to manage hardware resources in a data center using code instead of a manual process. It can automate the entire lifecycle of workloads running on data centers, including provisioning, deployment, and management. Different tools can be used to automate deployment, including Terraform, *Pulumi*, *AWS CloudFormation*, and *Puppet*. Due to its broad compatibility with cloud providers, we employ Terraform as our IaC in this work. Terraform employs the HashiCorp Configuration Language (HCL), similar to JSON but adding elements like variable declarations, loops, and conditionals.

The core functionality of Terraform is split into three main steps after the Terraform configuration is

Algorithm 1: Terraform Child Module Configuration.

Example of a **Child Module** configuration. Note that there are **optional** and some **required** variables.

```
module "app_deploy" {
    source = "./module/kubernetes_job" # Required
    app_path = "./my/path/toapp" # Required
    cpu_limit = 2 # Required
    build_command = "my build command" # Optional
    run_command = "my run command" # Optional
    workdir = "/usr/src/app" # Optional
    custom_image = "my-custom-image" # Optional
    kubeconfig_path = "my-kubeconfig-path" # Optional
}
```

written, as discussed next. (*i*) *Init*, which initializes the working directory, sets up the environment and prepares Terraform for operation; (*ii*) *Plan*, which creates an execution plan to let the users preview the changes that Terraform plans to make to the infrastructure; and (*iii*) *Apply*, where the actions proposed in the Terraform plan are executed. Also, in the end, the Destroy is responsible for destroying all remote objects managed by a particular Terraform configuration.

Terraform uses modules to manage infrastructures that scale effectively in terms of resources. A Terraform module is a collection of standard configuration files stored in a dedicated directory. These modules group together resources that serve a specific purpose, which helps reduce the amount of code developers need to write for similar infrastructure components. There are two types of modules in Terraform: root and child. The root module manages the overall setup, resources, and global settings. Child modules are reusable components used by the root or other child modules. When commands like init, plan, or apply are run, Terraform starts with the root module, loading its configurations and dependencies before moving to the child modules. Each child module includes specific resources like setting up a database, load balancer, or virtual network. An example of child module configuration is depicted in Algorithm 1, where the module is named "*app_deploy*. Within it, the configurations that must be used when deploying the workload for execution are defined, including the source for the module Kubernetes, the path for the workload (*app_path*), the limit of CPUs (*cpu_limit*) and optional commands needed by the workload.

2.3 Scalability of Parallel Workloads

Many studies indicate that maximizing available cores and cache does not guarantee optimal performance or energy efficiency for specific parallel workloads due to inherent hardware and software limitations (Suleman et al., 2008) (Subramanian et al., 2013). Workloads requiring frequent main memory access for private data encounter scalability issues as off-chip bus saturation limits performance (Suleman et al., 2008). While increased threads intensify bus demand, bandwidth is restricted by fixed I/O pin constraints (Ham et al., 2013), preventing proportional scaling and leading to elevated energy use without corresponding performance gains.

For shared data workloads, shared memory access frequency becomes critical as threads increase, impacting performance and energy. Inter-thread communication typically accesses distant memory regions, such as last-level caches or main memory, which incur greater latency and power consumption than private caches, introducing bottlenecks in execution (Subramanian et al., 2013). In synchronization, accessing shared variables requires sequential access to prevent race conditions, causing serialization that increases execution time and energy consumption within these critical sections (Suleman et al., 2008).

2.4 Related Work

Infrastructure as Code has gained attention in recent years due to its ability to automate the provisioning and management of infrastructure through code. Sandobalin et al., (Sandobalin et al., 2017) developed an infrastructure modeling tool for cloud provisioning to decrease the workload for development and operations teams. Borovits et al., (Borovits et al., 2020) propose DeepIaC, a deep learning-based approach for detecting linguistic anti-patterns in IaC through word embeddings and abstract syntax tree analysis. Vuppalapati et al., (Vuppalapati et al., 2020) discuss the automation of Tiny ML Intelligent Sensors DevOps using Microsoft Azure. Sandobalín et al., 2020 (Sandobalín et al., 2020) compare a model-driven tool (Argon) with a code-centric tool (Ansible) to evaluate their effectiveness in defining cloud infrastructure. Similarly, Palma et al., (Palma et al., 2020) propose a catalog of software quality metrics for IaC. Kumara et al., (Kumara et al., 2020) present a knowledge-driven approach for semantic detecting smells in cloud infrastructure code. Lepiller et al., (Lepiller et al., 2021) analyze IaC to prevent intra-update sniping vulnerabilities, showcasing the importance of leveraging tools for infrastructure configuration management. Saavedra et al., (Saavedra and Ferreira, 2022) introduce GLITCH, an automated approach for polyglot security smell detection in IaC.

2.4.1 Our Contributions

Based on the works discussed above, this paper makes the following contributions: (*i*) Unlike strategies that focus solely on optimizing the execution of parallel applications on a single-node machine by adjusting the TLP degree and other parameters (e.g., DVFS), our approach, *SmartNodeTuner*, offers a comprehensive solution. It not only identifies the best node to run the workload but also considers the optimal TLP degree for executing parallel workloads. Compared to existing solutions that leverage IaC to automate the setup of HPC environments, our strategy simplifies the end user or system administrator process by seamlessly and simultaneously addressing both the ideal node and TLP degree necessary for efficient workload execution.

3 SmartNodeTuner

In this section, we present *SmartNodeTuner*, our proposed approach. The primary objective of *SmartNodeTuner* is to optimize the balance between performance and energy consumption, as measured by the Energy-Delay Product (EDP) metric. This optimization applies to homogeneous and heterogeneous cloud environments while executing parallel workloads. To do that, *SmartNodeTuner* is divided into two main engines: *BuildModel* and *AutoDeploy*, as illustrated in Fig. 1. The *BuildModel* is responsible for training an ANN model and building a predictor, as discussed in Section 3.1. On the other hand, the *AutoDeploy* engine is responsible for automatically deploying workloads on cloud infrastructures using the built predictor and IaC, as described in Section 3.2

3.1 BuildModel Engine

To train and build the predictor that will be used by the *AutoDeploy* engine, the *BuildModel* is divided into two main steps: *feature extraction* and *model generation*, as illustrated in Fig. 1 and discussed next.

3.1.1 Extracting Features for the ANN Model

Given the training set composed of workload binaries provided by the user to train the ANN model ([1]), the first step of this engine is to collect the hardware and software metrics that will be used to train the ANN model. Then, *SmartNodeTuner* packages these workloads in Docker images before deploying them to execute across different architectures. During the DSE, each worker node runs each workload with the number of threads ranging from 1 to the number of available hardware threads. *SmartNodeTuner* does not employ thread oversubscription since it has demonstrated no performance and energy improvements in parallel workloads (Huang et al., 2021).

During execution, metrics for each combination of workload, worker node, and thread count are collected: **CPU Utilization**: Ranges from 0 to 1, measuring how effectively the threads use the cores. Values near 1.0 with maximum threads indicate good scalability; values near 0.0 suggest poor scalability. **Instructions Per Cycle** (IPC): Indicates the number of instructions executed per clock cycle. **Cache Memory Hit/Miss Rate**: Assesses data access efficiency in cache memory. These metrics determine if a workload is CPU- or memory-intensive. For example, a high cache miss rate and low CPU utilization may indicate that inter-thread communication hampers scalability. Additionally, *SmartNodeTuner* collects performance metrics such as execution time (in seconds), energy consumption (in joules), and calculates the EDP to determine the optimal worker node and thread count for each workload. Tools like AMDuProf for AMD processors and Intel VTune for Intel architectures collect these metrics directly from hardware counters.

At the end of this phase, *SmartNodeTuner* stores all collected data in its internal dataset, which includes: *workload description*, *worker node identifier*, *number of threads used*, *extracted features*, and *optimal configuration*. To ensure data integrity and prevent issues like overfitting or underfitting in the machine learning model, *SmartNodeTuner* applies Discretization and Min-Max Normalization to the dataset. Discretization converts categorical data into numerical values, and normalization scales all metric values to a standard range between 0 and 1, maintaining consistency across the dataset.

3.1.2 Generating the ANN Predictor Model

After preparing the dataset in the initial step, it is used to train the ANN predictor model. The ANN's input layer is designed to accept specific parameters, including the Workload ID, Worker Node, TLP degree, CPU utilization statistics, IPC, and Cache memory hit-and-miss rates. To maximize the ANN model's performance, *SmartNodeTuner* focuses on fine-tuning several critical hyperparameters. These include the number of hidden layers in the network, the number of neurons within each layer, the choice of activation function, the learning rate, the momentum parameter, and the total number of training epochs. To find the optimal hyperparameter value, *SmartNodeTuner* em-

[1] Available at omitted due to double-blind policy

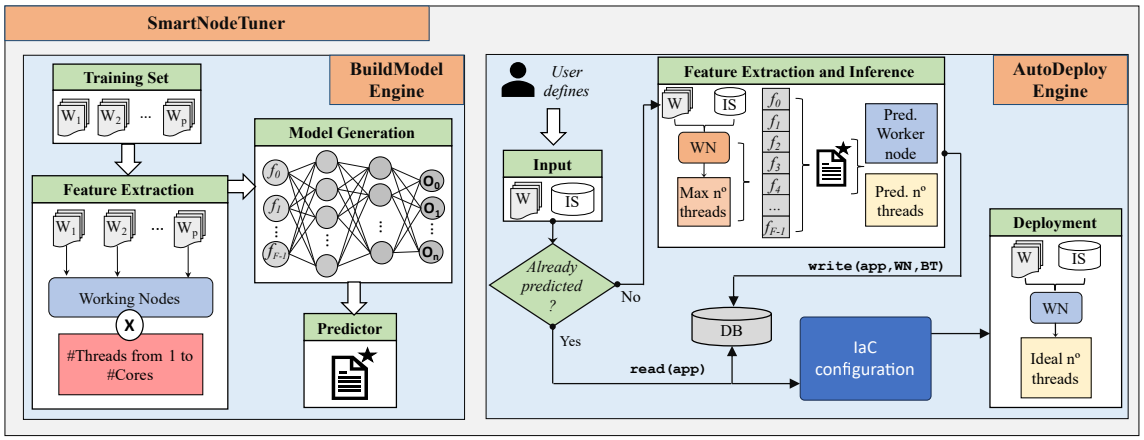


Figure 1: Workflow of each Engine used by *SmartNodeTuner*.

ploys *KerasTuner*, which automates the exploration of the parameter space, identifying the most effective combination of hyperparameters.

After determining the optimal hyperparameters, the ANN model is trained using the provided dataset and divided into training and testing subsets. To ensure robustness and generalizability of the model, we employ Stratified *k-Fold* cross-validation during the evaluation phase. This technique is suitable for datasets with class imbalance, as it preserves the proportion of classes in each fold. The dataset is then partitioned into *k* stratified folds; in each iteration, one fold serves as the validation set while the remaining *k-1* folds constitute the training set. *SmartNodeTuner* performs 20 iterations of this cross-validation process. Ultimately, the predictor model is selected based on the highest accuracy across all iterations.

3.2 AutoDeploy Engine

Given the predictor model built in the last step (which is only performed once), the *AutoDeploy* engine is responsible for managing the workload execution on the environment, as shown in Fig. 1. For that, it predicts ideal combinations of worker node and TLP degree and then uses these values to launch the workload via IaC transparently.

3.2.1 Predicting Ideal Combinations

The execution phase begins when the user provides the workload binary and input set encapsulated into a container for execution in the cluster environment. This input prompts *SmartNodeTuner* to utilize the trained ANN model to generate recommendations to optimize the EDP. It is worth mentioning that although *SmartNodeTuner* is configured to optimize the EDP of applications, it can be modified to optimize the workload for other metrics like performance or energy. Then, *SmartNodeTuner* checks its internal database to determine whether the workload has been previously executed by comparing the hash information. When it is the first time the workload is executed on the system, *SmartNodeTuner* performs the following operations. (*i*) The container is configured to run on any available worker node with the TLP degree equal to the hardware threads available. (*i*) During execution, *SmartNodeTuner* collects the same hardware and software metrics as the *Build-Model Engine*. (*iii*) The collected metrics are pre-processed using discretization and normalization techniques to prepare the data for input into the predictor model. (*iv*) The pre-processed data is fed into the predictor model, which predicts an ideal worker node and thread count. (*v*) The predicted configuration is then stored in the database and associated with the workload details to facilitate quick retrieval in future executions. On the other hand, if the workload has been executed and predicted before, *SmartNodeTuner* retrieves the stored predicted configurations from the database and moves to the *IaC configuration*.

3.2.2 Automating Workload Deployment via IaC

In this stage, the IaC configuration module orchestrates the deployment of containers with workloads according to the ideal node and TLP degree combination determined in the previous stage. This configuration operates within a Kubernetes v.1.30 cluster, while the module was implemented using *Terraform v1.9.0*, leveraging IaC to automate resource allocation processes.

The deployment begins with *SmartNodeTuner* using specific Terraform data source blocks to request the Kubernetes API via the official provider for node availability and assess the following resource metrics across the Kubernetes cluster: available CPU cores

and memory capacity from each worker node to confirm that nodes meet the necessary workload requirements. With this information, *SmartNodeTuner* attempts to allocate the workload on the previously recommended node. If this node is unavailable, the module automatically examines other nodes in the cluster to find the next best match, considering the proximity of the number of CPUs to the ideal TLP degree predicted by the ANN model.

Deployment specifications are abstracted from the user side, as they are automatically populated by the information populated in the Child Module declared in Terraform, including the worker node ID, the container path for the workload binary, and the number of CPU cores to be allocated. *SmartNodeTuner* configures this file with Kubernetes-specific API directives such as *cpuRequests* and *cpuLimits* within the pod specifications; the job is then applied to the cluster by defining the Terraform *kubernetes_job* resource in the root module, which triggers the Kubernetes API to instantiate the container based on the specifications provided. Thanks to the automated configurations of the IaC module, the user will not need to deal with Kubernetes *.yaml* files or manual commands via the command line. The module itself will be in charge of taking the application to the target node and deploying it with the correct configurations. Although in this work, *SmartNodeTuner* was developed and configured to operate with a Kubernetes cluster, the IaC configuration module's design is extendable to support other cloud environments, such as AWS Elastic Kubernetes Service (EKS), Google Kubernetes Engine (GKE), and Microsoft Azure.

4 METHODOLOGY

4.1 Execution Environment

We conducted our experiments within a private cloud environment featuring a variety of hardware configurations. This setup included a master node responsible for distributing applications to worker nodes, as well as three worker nodes, each with distinct processing capabilities: ***WN16*** – AMD Ryzen 7 2700, with 16 HW threads and 32GB RAM; ***WN24*** – AMD Ryzen 2920X, with 24 HW threads and 96GB RAM; and ***WN64*** AMD Threadripper 3990X, with 64 HW threads and 128GB RAM. Every node ran the Debian OS, Kubernetes version 1.30, and Docker version 23.0. The applications were compiled using GCC/G++ version 12.0 with the optimization flag -O3.

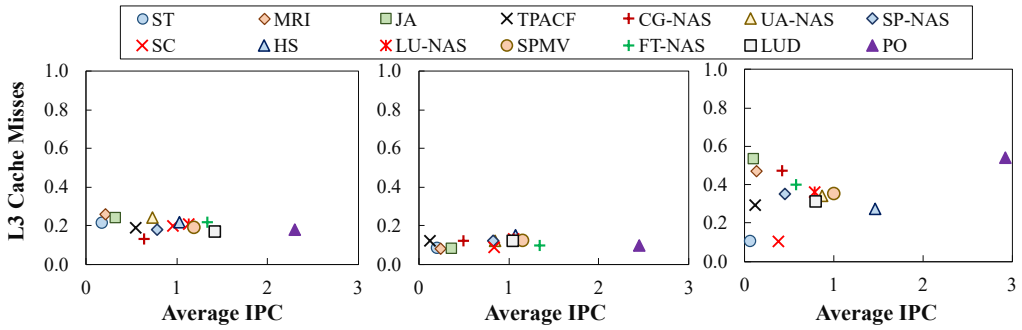
4.2 Parallel Workloads

We employed a set of twenty-four workloads already parallelized and written in *C* and *C++*. These workloads were categorized into training datasets and validation datasets.

Training DataSet: For the training phase, we selected fourteen workloads with different characteristics of L3 cache miss ratio and average number of instructions per cycle (IPC), as shown in Fig. 2: **Three** applications from the Rodinia Benchmark Suite (Che et al., 2009): hotspot (HS), lower-upper decomposition (LUD), and streamcluster (SC). **Five** kernels and pseudo-applications from the NAS Parallel Benchmarks (Bailey et al., 1991): CG, FT, LU, SP, and UA. **Three** applications from various other domains: the Jacobi method (JA), the Poisson equation solver (PO), and the STREAM benchmark (ST). **Three** applications from the Parboil Benchmark Suite (Stratton et al., 2012): MRI, SPMV, and TPACF.

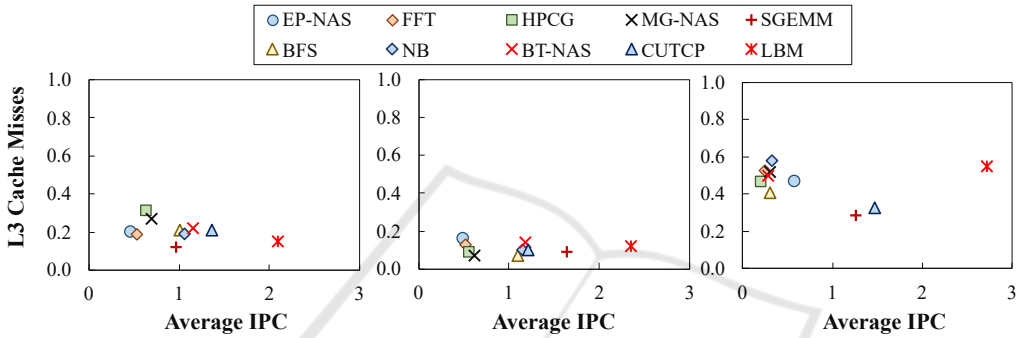
Validation DataSet: To validate *SmartNodeTuner*, we selected ten applications that exhibit varying characteristics in terms of CPU and memory usage, as detailed in Fig. 3: **Four** from the Parboil Benchmark suite: BFS, CUTCP, LBM, and SGEMM. **Three** from the NAS Parallel Benchmark suite: BT, EP, and MG. **Three** from other domains: FFT, HPCG, and NB. The applications can also be categorized based on their degree of parallelism, as measured using *AMD uProf*. *Low parallelism*: limited scalability due to inherent constraints in their computational structure or workload distribution, and include *BFS*, *FFT*, *HPCG*, *NB*, and *SGEMM*. *Medium parallelism*: moderate scalability, leveraging parallel resources more effectively than low-parallelism workloads but not fully exploiting the available threads. Examples are *LBM* and *MG-NAS*. *High parallelism*: These applications efficiently scale across multiple threads, fully utilizing the parallel capabilities of the hardware. Examples include *BT-NAS* and *CUTCP*.

We have chosen these applications because of their diversity in computational and memory access patterns, which mirror real-world parallel cloud workloads. The training dataset includes applications with varied L3 cache miss ratios and IPC, such as those from the Rodinia and NAS-PB suites, enabling the evaluation of our proposed framework under different hardware utilization scenarios. Similarly, the validation dataset includes applications with distinct CPU and memory usage behaviors, ranging from compute-bound tasks like LBM to memory-intensive applications like HPCG, ensuring comprehensive coverage of cloud-specific challenges such as resource allocation and heterogeneity.



a) AMD-16 b) AMD-24 c) AMD-64

Figure 2: Behavior of each workload used to train the model employed by *SmartNodeTuner*.



a) AMD-16 b) AMD-24 c) AMD-64

Figure 3: Behavior of each workload used to validate *SmartNodeTuner*.

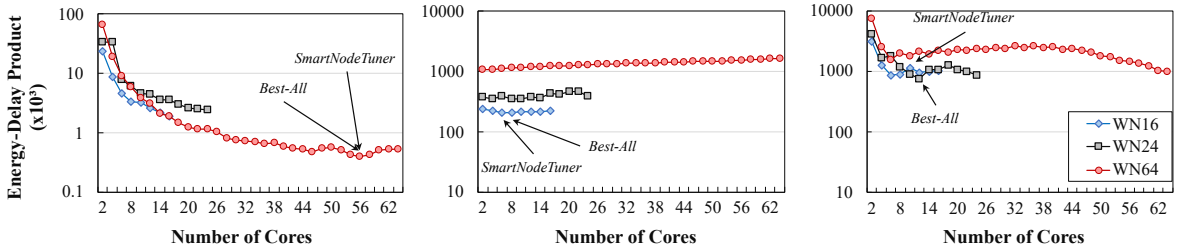
5 EXPERIMENTAL EVALUATION

In this section, we discuss the results of employing *SmartNodeTuner* to execute parallel workloads on a private heterogeneous cloud. To assess the effectiveness of our approach, we compared its results against distinct scenarios:

STD-WN16, **STD-WN24**, and **STD-WN64**: each workload was executed on the respective worker node with the number of threads that matches the number of cores (e.g., 16, 24, and 64, respectively), which is the standard practice employed to execute parallel workloads. **Best-WN16**, **Best-WN24**, and **Best-WN64**: In this scenario, we conducted a thorough search to determine the optimal number of threads that achieved the best EDP for each node. Each configuration represents the execution of the workload using the optimal number of threads on each worker node. **Random**: a method where the workloads are randomly assigned to worker nodes. **Kube-Scheduler**: This scenario uses Kubernetes' built-in scheduling component. **Best-All**: an ideal scenario where each application was executed with the best possible configuration regarding worker node selection and thread count, resulting in the lowest energy-delay product. This optimal configuration was identified by exhaustively testing all combinations of worker nodes, and thread counts for each workload.

5.1 Accuracy of *SmartNodeTuner*

Table 1 compares the configurations predicted by *SmartNodeTuner* with those identified through exhaustive search (referred to as **Best-All**) for the ten validation workloads. Each configuration is represented as $< worker\ node - \#threads >$. The table also indicates the rank of *SmartNodeTuner*'s prediction among all possible configurations. As shown, no single configuration (working node and number of threads) provides the best trade-off between performance and energy consumption across all applications. For instance, the optimal configuration found by **Best-All** for *BFS* is to run it with four threads on the *WN16* system, whereas the *CUTCP* benchmark performs best with 56 threads on the working node with 64 cores. To further analyze this behav-



a) CUTCP b) FFT c) LBM

Figure 4: EDP behavior of three workloads when running on the evaluated platforms.

Table 1: Combinations found by **Best-All** and *SmartNodeTuner* for each workload.

	Best-All	SmartNode Tuner	Top-Best (%)	EDP Diff
BFS	WN16-4	WN16-6	< 2%	2.05
BT-NAS	WN64-52	WN64-64	< 8%	1.25
CUTCP	WN64-56	WN64-56	< 1%	1.00
EP-NAS	WN16-16	WN16-16	< 1%	1.00
FFT	WN16-8	WN16-6	< 2%	1.01
HPCG	WN16-8	WN16-8	< 1%	1.00
LBM	WN24-12	WN16-14	< 7%	1.31
MG-NAS	WN24-12	WN24-12	< 1%	1.00
NB	WN16-4	WN16-2	< 2%	1.77
SGEMM	WN16-8	WN16-8	< 1%	1.00

ior, Figure 4 illustrates the EDP for all evaluated configurations of working nodes and thread counts when running three applications with distinct characteristics (*CUTCP*, *FFT*, and *LBM*). We also highlight the configurations found by **Best-All** and predicted by *SmartNodeTuner*.

For applications with a high average IPC and a low ratio of time spent accessing main memory (indicated by fewer L3 cache misses), the competition for shared resources is reduced. In such cases, running these applications on a working node with more cores results in significant EDP reductions during execution. This behavior is evident in the *CUTCP* application, as shown in Figure 4.a, where the application scales well and benefits from the large number of cores and cache memory available on the WN64 system. A similar pattern was observed for *BT-NAS*.

Conversely, for applications with limited parallelism and a moderate ratio of time spent accessing main memory, the best EDP results are achieved by running them on a working node with fewer cores, minimizing the impact of data communication among threads. This was the case for applications such as *BFS*, *EP-NAS*, *FFT*, *HPCG*, *NB*, and *SGEMM*. For instance, Figure 4.b illustrates the scenario for the *FFT* application, where the *WN16* system delivered the best results. Additionally, applications with a moderate degree of TLP achieved optimal performance on the working node with 24 cores, as observed in the *LBM* application shown in Figure 4.c.

Analyzing the results obtained by *SmartNodeTuner* in Table 1, it correctly predicted the optimal configuration in half of the cases. While this accuracy rate may appear low, it underscores the complexity of the optimization challenge, with 104 possible configurations per workload. On top of that, 80% of its predictions were within the *Top-2* configurations, and all were within the *Top-8*. Table 1 also compares the EDP between *SmartNodeTuner* and **Best-All** (*EDP Diff* column), with values normalized to the **Best-All** results. Hence, a value close to 1.0 means that *SmartNodeTuner* reaches a configuration near the optimal. In almost all cases, *SmartNodeTuner* predicted the combination of worker node and number of threads within the 2% of best solutions, leading to a difference of only 19% of EDP across all workloads. The worst case for *SmartNodeTuner* was for the *NB* and *BFS* workloads due to the sensitivity of these applications to thread synchronization issues. For this type of application, increasing the number of active threads leads to more time spent synchronizing data within parallel regions, which can degrade performance and energy efficiency.

This behavior is illustrated for the *BFS* application in Figure 5 on the working node with 16 cores (*WN16*). The *x-axis* represents the number of active threads. At the same time, the execution time is divided into two parts: the time spent executing the parallel region and the time spent synchronizing data. Therefore, the total execution time is the sum of these parts. The secondary *y-axis* shows the total energy consumption, measured in Joules. As depicted, the execution time decreases as the number of threads increases from one to four. However, beyond this point, synchronization overhead surpasses the execution time of the benefits obtained due to parallelization, resulting in increased execution time and energy consumption, thereby worsening the EDP. Although *SmartNodeTuner* was able to predict a near-optimal configuration (WN16-6 instead of WN16-4), the EDP difference compared to the *Best-All* solution

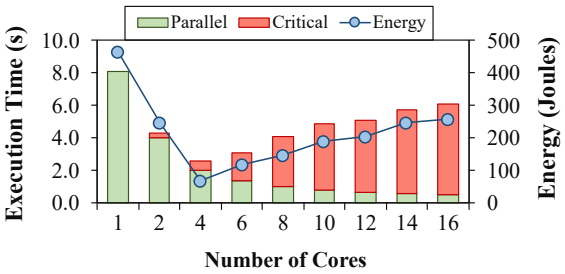


Figure 5: Thread Scalability of BFS on the WN16.

was 2.05 times.

5.2 EDP Comparison

In this subsection, we compare the EDP results of each strategy running the workloads on the target private cluster, as described in Section 4. For that, Fig. 6 illustrates the EDP of each strategy normalized to the *Best-All* for each workload, represented by the ***black line***. Moreover, Fig. 7 depicts the distribution of the EDP results normalized to the best EDP achieved on each workload (*Best-All*). Hence, the closer the values are to 1.0, the better the EDP. In this analysis, our primary interest is achieving a distribution of EDP results on the validation workloads as close as possible to the *Best-All*. Hence, an ideal outcome would be a compact *boxplot* in Fig. 7, indicating low variability in achieving the best EDP for each workload and near 1.0.

We begin by analyzing the EDP of our strategy, *SmartNodeTuner*, compared to the standard execution strategy on each worker node (*STD-WN16*, *STD-WN24*, and *STD-WN64*). As shown in Fig. 6, *SmartNodeTuner* achieved better EDP across most cases. The most significant gains by choosing an ideal worker node and TLP degree were observed in applications with limited thread scalability due to data synchronization overhead, such as *NB* and *BFS*. As discussed by Suleman et al., (Suleman et al., 2008; Lorenzon et al., 2018; Maas et al., 2024), using the maximum number of threads to execute this kind of workload increases execution time and energy usage due to the overhead on the critical regions, negatively impacting EDP. On the other hand, in scenarios where ideal EDP aligns with maximum thread count, the results were similar (e.g., EP-NAS for the *STD-WN16*. Overall, *SmartNodeTuner* achieved EDP improvements, with geometric means showing enhancements of 54.9%, 77.8%, and 81.7% on *STD-WN16*, *STD-WN24*, and *STD-WN64* configurations, respectively. Even when compared to the best EDP achieved per worker node (*Best-WN16*, *Best-WN24*, and *Best-WN64*), *SmartNodeTune* achieves better overall EDP, highlighting the importance of selecting not only the optimal thread count per worker node but also finding an ideal worker node to execute the given workload. On average, across all workloads, *SmartNodeTuner* improves EDP by 17.9%, 35.1%, and 43.4% over the best threading configuration on the machines, respectively.

While *Random* and *Kube-Scheduler* can deliver better EDP results than the standard execution on each worker node, neither outperforms the EDP improvements provided by *SmartNodeTuner*. On average, *SmartNodeTuner* achieves a 38.2% higher EDP efficiency than *Kube-Scheduler* across all applications. The main reason we found during the experiments is that the decisions made by the scheduler do not consider the efficiency in resource utilization, leading to less optimal choices for node and thread allocation. Instead, it considers resource availability, e.g., CPU and memory behavior. Differently, *SmartNodeTuner* considers the workload characteristics regarding resource efficiency when deploying it for execution, optimizing thread distribution, and node allocation.

Finally, let us consider the EDP distribution across configurations, shown in Fig. 7. The goal here is to achieve a compact distribution near 1.0, indicating both low variability and a high EDP efficiency relative to the exhaustive search results (*Best-All*). In this context, *SmartNodeTuner* maintained a consistently narrow distribution, centered close to 1.0 across various workloads. By contrast, the other configurations have wider spreads and higher median values, reflecting more significant inconsistency and generally worse EDP efficiency overall.

5.3 Impact on the Performance and Energy Consumption

Improving, at the same time, the energy efficiency and performance in cloud computing environments is challenging as it requires balancing both metrics so that one metric is not compromised due to the improvements on the other. In this scenario, to assess the efficacy of *SmartNodeTuner* in achieving this balance, we compared the performance and energy consumption to all the previously discussed strategies. For that, Fig. 8a shows the performance reached by each strategy normalized to the *Best-All* configuration, considering the geometric mean of all workloads. In this plot, the closer the value is to 1.0, the better the performance. Similarly, Fig. 8b depicts the energy consumption normalized to the best result. In this plot, the lower the value is, the less energy was spent during execution.

Because our approach, *SmartNodeTuner*, can predict configurations that are most of the time within

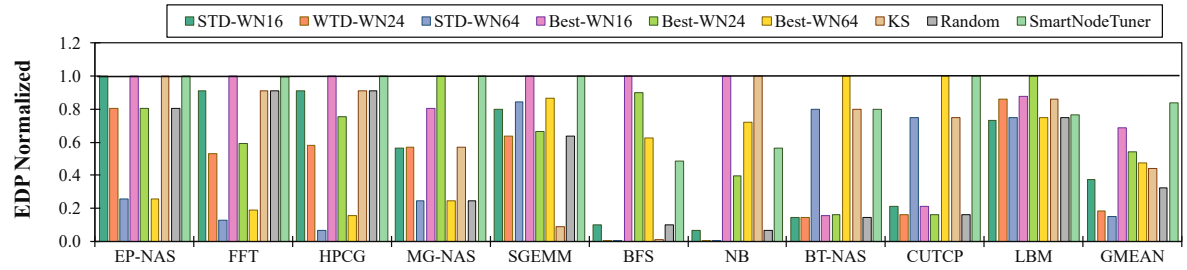


Figure 6: EDP results on each workload, normalized to the *Best-All*, represented by the black line.

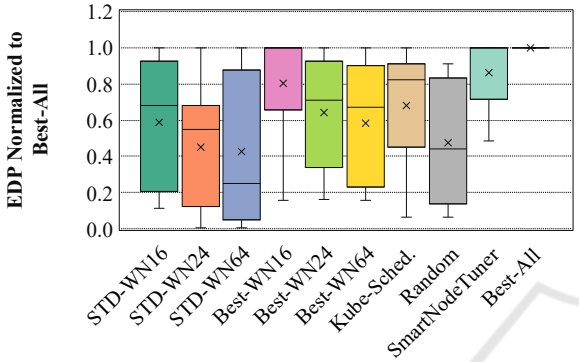


Figure 7: Distribution of EDP results for each strategy across all workloads.

the Top-2%, it can reach energy consumption levels as close to the ideal one (only 7.01% of difference) while not jeopardizing the overall performance (10.5%) as the other strategies do. When comparing *SmartNodeTuner* with the configuration that delivers the lowest overall energy consumption (*Best-WN16*), it is 5.3% more energy-hungry but reaches performance levels 28.2% higher. When only the performance matters, *Best-WN24* can deliver better performance without considering the *Best-All* configuration (6% higher than *SmartNodeTuner*) at the price of 64.6% more energy spent.

5.4 Overhead of *SmartNodeTuner*

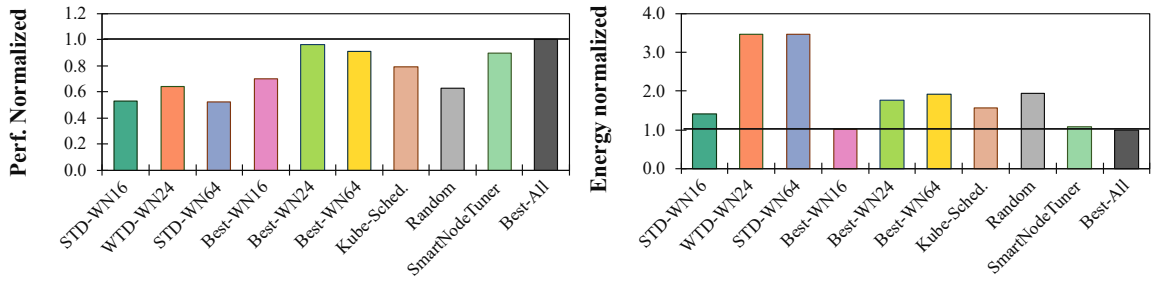
Achieving configurations consistently within the Top-2% best solutions allows *SmartNodeTuner* to approximate the EDP efficiency of an exhaustive search (*Best-All*) with much lower overhead. Unlike exhaustive search, *SmartNodeTuner* profiles each application only once with a default configuration, and the inference process takes only 0.0093s per lookup. In this scenario, the time it took for *SmartNodeTuner* to run each target application with the standard configuration and predict an ideal combination of TLP degree and worker node was only *413.75s*, compared to *26377.01s* of the exhaustive search. On the other hand, the feature extraction part for the ANN model incurs the highest computational cost: 2.38 hours on *WN16*, 2.79 hours on *WN24*, and 10.31 hours on *WN64*, with respective energy costs of $3.42x10^5$ J, $8.61x10^5$ J, and $3.50x10^6$ J. However, it is worth mentioning that this extraction phase is performed only once, and this cost can be further minimized via strategies like sampling, reduced input sets, or distributed computing, which are not the goal of this paper.

6 CONCLUSION

We have presented *SmartNodeTuner*, a framework for optimizing the performance and energy consumption when executing HPC workloads in cloud environments using AI and IaC. It considers the behavior of parallel workloads to predict ideal combinations of worker nodes and TLP degrees. By incorporating IaC into the automation process of *SmartNodeTuner*, the resource management is simplified, being applied to diverse cloud infrastructures. When evaluating *SmartNodeTuner* over the execution of ten well-known parallel workloads on a heterogenous environment, we show that it predicts combinations that reach EDP values close to the ones achieved by the exhaustive search, improving the EDP by 38.2% compared to the standard scheduler used by Kubernetes. We also show that by employing *SmartNodeTuner*, the application's performance is marginally affected while providing significant energy savings. As future work, we plan to increase the compatibility of *SmartNodeTuner* with other cluster orchestrators, allowing users more flexibility in selecting cloud and HPC solutions.

ACKNOWLEDGEMENTS

This study was partly financed by the CAPES - Finance Code 001, FAPERGS - PqG 24/2551-0001388-1, and CNPq.



(a) Performance (b) Energy consumption

Figure 8: Performance and Energy results for each strategy normalized to *Best-All*.

REFERENCES

Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., Simon, H. D., Venkatakrishnan, V., and Weeratunga, S. K. (1991). The nas parallel benchmarks and summary and preliminary results. In *ACM/IEEE SC*, pages 158–165, USA. ACM.

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177.

Borovits, N., Kumara, I., Krishnan, P., Palma, S. D., Di Nucci, D., Palomba, F., Tamburri, D. A., and van den Heuvel, W.-J. (2020). Deepiac: deep learning-based linguistic anti-pattern detection in iac. In *Proceedings of the 4th ACM SIGSOFT International Workshop on Machine-Learning Techniques for Software-Quality Evaluation*, MaLTeSQuE 2020, page 7–12, New York, NY, USA. Association for Computing Machinery.

Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. (2009). Rodinia: A benchmark suite for heterogeneous computing. In *IEEE Int. Symp. on Workload Characterization*, pages 44–54, DC, USA. IEEE Computer Society.

Ham, T. J., Chelepalli, B. K., Xue, N., and Lee, B. C. (2013). Disintegrated control for energy-efficient and heterogeneous memory systems. In *IEEE HPCA*, pages 424–435.

Huang, H., Rao, J., Wu, S., Jin, H., Jiang, H., Che, H., and Wu, X. (2021). Towards exploiting cpu elasticity via efficient thread oversubscription. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '21, page 215–226, New York, NY, USA. Association for Computing Machinery.

Kumara, I., Vasileiou, Z., Meditskos, G., Tamburri, D. A., Heuvel, W.-J. V. D., Karakostas, A., Vrochidis, S., and Kompatsiaris, I. (2020). Towards semantic detection of smells in cloud infrastructure code. *ARXIV-CS.SE*.

Lepiller, J., Piskac, R., Schäf, M., and Santolucito, M. (2021). Analyzing infrastructure as code to prevent intra-update sniping vulnerabilities. *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*.

Liu, F., Tong, J., Mao, J., Bohn, R., Messina, J., Badger, L., and Leaf, D. (2012). *NIST Cloud Computing Reference Architecture: Recommendations of the National Institute of Standards and Technology*. CreateSpace Independent Publishing Platform, USA.

Lorenzon, A. F. and Beck Filho, A. C. S. (2019). *Parallel computing hits the power wall: principles, challenges, and a survey of solutions*. Springer Nature.

Lorenzon, A. F., De Oliveira, C. C., Souza, J. D., and Beck, A. C. S. (2018). Aurora: Seamless optimization of openmp applications. *IEEE transactions on parallel and distributed systems*, 30(5):1007–1021.

Maas, W., de Souza, P. S. S., Luizelli, M. C., Rossi, F. D., Navaux, P. O. A., and Lorenzon, A. F. (2024). An ann-guided multi-objective framework for power-performance balancing in hpc systems. In *Proceedings of the 21st ACM International Conference on Computing Frontiers*, CF '24, page 138–146, New York, NY, USA. Association for Computing Machinery.

Márquez, G., Villegas, M. M., and Astudillo, H. (2018). A pattern language for scalable microservices-based systems. In *ECSA*, NY, USA. ACM.

Masanet, E., Shehabi, A., Lei, N., Smith, S., and Koomey, J. (2020). Recalibrating global data center energy-use estimates. *Science*, 367(6481):984–986.

Navaux, P. O. A., Lorenzon, A. F., and da Silva Serpa, M. (2023). Challenges in high-performance computing. *Journal of the Brazilian Computer Society*, 29(1):51–62.

Palma, S. D., Nucci, D. D., and Tamburri, D. A. (2020). Ansiblemetrics: A python library for measuring infrastructure-as-code blueprints in ansible. *SOFTWAREX*.

Saavedra, N. and Ferreira, J. F. (2022). Glitch: Automated polyglot security smell detection in infrastructure as code. *ARXIV-CS.CR*.

Sandobalin, J., Insfrán, E., and Abrahão, S. M. (2017). An infrastructure modelling tool for cloud provisioning. *IEEE International Conference on Services Computing (SCC)*.

Sandobalín, J., Insfran, E., and Abrahão, S. (2020). On the

effectiveness of tools to support infrastructure as code: Model-driven versus code-centric. *IEEE ACCESS*.

Stratton, J., Rodrigues, C., Sung, I., Obeid, N., Chang, L., Anssari, N., Liu, G., and Hwu, W. (2012). Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*.

Subramanian, L., Seshadri, V., Kim, Y., Jaiyen, B., and Mutlu, O. (2013). MISE: Providing performance predictability and improving fairness in shared main memory systems. In *IEEE HPCA*, pages 639–650.

Suleman, M. A., Qureshi, M. K., and Patt, Y. N. (2008). Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on cmps. *SIGARCH Comput. Archit. News*, 36(1):277–286.

Vuppalapati, C., Ilapakurti, A., Chillara, K., Kedari, S., and Mamidi, V. (2020). Automating tiny ml intelligent sensors devops using microsoft azure. *IEEE International Conference on Big Data (Big Data)*.