Energy-efficient Application Resource Scheduling using Machine Learning Classifiers

Connor Imes*
University of Chicago
ckimes@cs.uchicago.edu

Steven Hofmeyr
Lawrence Berkeley National Laboratory
shofmeyr@lbl.gov

Henry Hoffmann
University of Chicago
hankhoffmann@cs.uchicago.edu

ABSTRACT

Resource scheduling in high performance computing (HPC) usually aims to minimize application runtime rather than optimize for energy efficiency. Most existing research on reducing power and energy consumption imposes the constraint that little or no performance loss is allowed, which improves but still does not maximize energy efficiency. By optimizing for energy efficiency instead of application turnaround time, we can reduce the cost of running scientific applications. We propose using machine learning classification, driven by low-level hardware performance counters, to predict the most energy-efficient resource settings to use during application runtime, which unlike static resource scheduling dynamically adapts to changing application behavior. We evaluate our approach on a large shared-memory system using four complex bioinformatic HPC applications, decreasing energy consumption over the naive *race* scheduler by 20% on average, and by as much as 38%. An average increase in runtime of 31% is dominated by a 39% reduction in power consumption, from which we extrapolate the potential for a 24% increase in throughput for future over-provisioned, power-constrained clusters. This work demonstrates that low-overhead classification is suitable for dynamically optimizing energy efficiency during application runtime.

CCS CONCEPTS

• **General and reference → Performance**; • **Hardware → Power estimation and optimization**; • **Software and its engineering → Power management**;

KEYWORDS

Energy Efficiency, Power Management, Runtime Control, Machine Learning

ACM Reference Format:
Connor Imes, Steven Hofmeyr, and Henry Hoffmann. 2018. Energy-efficient Application Resource Scheduling using Machine Learning Classifiers. In *ICPP 2018: 47th International Conference on Parallel Processing, August 13–16, 2018, Eugene, OR, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3225058.3225088

*Part of this research was conducted while Connor Imes was a Computing Sciences Summer Student at Lawrence Berkeley National Laboratory.

ICPP 2018, August 13–16, 2018, Eugene, OR, USA

ACM ISBN 978-1-4503-6510-9/18/08…$15.00
https://doi.org/10.1145/3225058.3225088

1 INTRODUCTION

Energy and power consumption will be first-class constraints in future exascale systems [46]. For example, these systems are predicted to have strict operating budgets of approximately 20 MW [26]. Additionally, exascale operating systems are required to actively "decrease the cost per scientific insight" [5].

A tremendous amount of prior work maximizes HPC performance, which directly addresses the cost—in time—of scientific discovery. Recent work has demonstrated (both in theory and in practice) that allocating maximum resources to an application is never more energy-efficient than a strategy that intelligently tailors resource usage to a particular application's needs [18, 24]. Thus, several projects further reduce the costs of science by reducing energy consumption while maintaining application performance [7, 22, 42]. Very little work, however, has explored more aggressively minimizing energy consumption. Minimizing application energy, even if runtime is increased, can (1) greatly reduce the cost (in Joules) of scientific insight and (2) increase application throughput in over-provisioned, power-constrained systems—such as future exascale computers [39, 43]—by allowing more applications to execute on a cluster simultaneously.

Minimizing energy in exascale systems is challenging because of the complexity of exascale compute nodes and the dynamic nature of applications, whose resource needs vary during execution. Within a node, many different resources can be tuned that impact computation efficiency, including dynamic voltage and frequency scaling (DVFS), multiple socket and core allocation, and the use of HyperThreads. The most energy-efficient configuration of these resources varies for different applications, and even within a single application as it transitions through multiple phases.

We propose to dynamically monitor application and system behavior at runtime and use machine learning classification to predict the most energy-efficient resource settings as the application executes. The proposed approach has two distinct benefits. First, it can be applied to new applications without modifying either the classifier or application, as it does not require application-level instrumentation or hooks. Second, classification's low overhead makes it suitable for running in-situ on compute nodes.

We first evaluate 15 different classification techniques and test their ability to correctly recall energy-efficient settings of 21 common HPC benchmarks and co-design applications on a quad-socket x86 compute node. We find that the complexity of the underlying system makes many of these classifiers unsuitable for the problem of predicting energy-efficient settings, but a handful are promising. We then choose 5 of the original 15 classifiers and evaluate their ability to dynamically manage four high-performance genome assembly

applications. These represent some of the most computationally-intensive bioinformatics applications, the most commonly-used of which only run on single shared-memory nodes. Consequently, there are HPC systems, such as NERSC's Genepool [36], that are predominantly used to concurrently execute many single-node applications, such as genome assemblers and comparative analysis [11].

Our evaluation provides the following insights. First, it is important to choose a classifier that can handle the complex, non-linear mapping of observed runtime behavior to energy-efficient system settings. Compared to a strategy that is required to maintain maximum performance, a poor classifier can cause a non-trivial increase in energy consumption, *e.g.*, by 22% in one case we explored. Given a sufficiently powerful classifier, however, the proposed approach reduces energy consumption by an average of 20%—only 5.4% higher energy than an offline oracle. We calculate that scaling this behavior to an over-provisioned, power-constrained cluster could increase total throughput by 24%. All of these results are achieved with no prior knowledge of the applications and no application-level modifications. We conclude that dramatic increases in scientific insight per Joule are possible by deploying machine learning classifiers to dynamically adjust resource usage.

In summary, this paper makes the following contributions:

(1) Proposes optimizing *energy efficiency* instead of *runtime* to decrease the cost of scientific computation.
(2) Establishes the problem complexity—of 15 different machine learning classification techniques evaluated, only some are suitable for optimizing energy efficiency.
(3) Demonstrates that sufficiently powerful classifiers can dramatically reduce energy consumption by accurately predicting energy-efficient system settings at runtime.

2 ON MAXIMIZING ENERGY EFFICIENCY

In this section, we first describe how maximizing energy efficiency is distinct from maximizing performance or minimizing power consumption. As such, finding the most energy-efficient system setting requires new techniques. We then discuss the challenges of learning a model for mapping application/system behavior to energy-efficient system settings.

2.1 Energy Efficiency is a Unique Challenge

Cluster energy efficiency can be defined simply as the ratio of applications completed per Joule of energy used. If each application execution represents some unit of scientific insight, then this metric expresses the cost of scientific insights in terms of operational costs (energy consumption). Thus, maximizing energy efficiency directly reduces the cost of scientific insight by minimizing application energy consumption. However, reducing energy consumption is more difficult than simply reducing power, since energy is the multiple of time and power, and it is hard to improve one without sacrificing the other. For example, raising processor speed typically reduces application runtime, but the requisite increase in power results in increased energy consumption. In short, maximizing energy efficiency *does not* mean using as little power as possible—it requires finding the optimal tradeoff between execution time and power consumption.

Recent energy-aware approaches reduce energy while maintaining application performance [22, 32, 42]. As these approaches do not trade application speed, they reduce energy strictly by reducing power. While it is tempting to use the same approaches to minimize energy by simply removing the constraint on application runtime, failing to account for the impact of lower-power resource settings on execution time can result in worse energy consumption.

Typical resource management approaches in HPC clusters primarily attempt to optimize application completion time, only worrying about power consumption to the extent that the total cluster power constraint is not violated, and ignoring energy consumption altogether. Although there are various techniques for assigning jobs to nodes, minimizing their runtime is usually achieved by running the individual compute nodes as fast as possible—an approach which we call *race*. Using *race* makes resource scheduling on a node easy, but it is never as energy-efficient as a more intelligent approach that understands how system settings affect performance and power tradeoffs on a per-application basis [24]. The reason is that *racing* to finish a job as fast as possible minimizes execution time, but the increase in power dwarfs the time savings, resulting in high energy consumption.

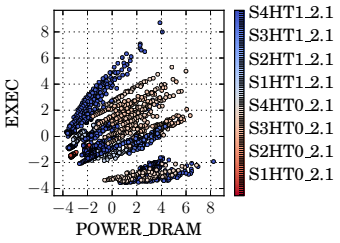
In addition to reducing costs, maximizing energy efficiency will also improve overall system throughput in power-constrained environments. In over-provisioned clusters, the hardware can draw more total power than the infrastructure can physically deliver to the system if nodes operate in the *race* setting [43]. Optimizing energy efficiency increases total cluster throughput by (1) allowing more applications to run in parallel due to lower per-application power consumption, while (2) still considering application runtime due to its impact on energy consumption. Therefore, new resource management approaches are required that focus solely on minimizing energy to reduce the cost of science (in Joules), even if the overall runtime is increased.

2.2 Learning Energy Efficiency

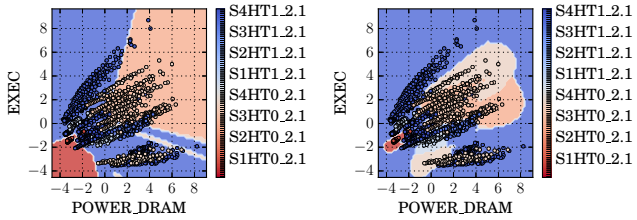
Identifying energy-efficient combinations of resource settings is challenging as there is no universal best setting for a system—it depends on the application and its configuration, even varying for different inputs. Even in a parallel system with homogeneous nodes and perfectly uniform application behavior, optimal settings can vary dramatically due to manufacturing variation [1]. Furthermore, many applications progress through different *phases* during execution. For example, an application may transition between compute- and memory-intensive processing, causing the most energy-efficient setting to change during runtime. Therefore, it is not optimal to choose a single setting statically when launching the application, necessitating a dynamic approach instead.

In this paper, we explore learning approaches that adjust system settings to minimize application energy. Specifically, we tune socket allocation, the use of HyperThreads, and processor DVFS. The learning component picks settings for the combination of these resources such that, at any point during the application execution, the system is operating in its most energy-efficient state.

Our goal is for the learner to find the most energy-efficient setting without requiring any application-level changes. Therefore, we use existing hardware performance counters as *features*. The



(a) Training data only.



(b) SVM (Linear kernel), recall=0.456. (c) SVM (RBF kernel), recall=0.710.

Figure 1: Training data and learned decision boundaries for two SVM classifiers using two primary features.

learner then determines a function that maps some subset of these features into the most energy-efficient system setting. However, system settings in modern compute nodes have very complicated interactions, so it is essential to use learning mechanisms that can produce accurate mappings despite this complexity.

We briefly illustrate this complexity using just two features—the performance counters `POWER_DRAM` (a measure of memory usage) and `EXEC` (a measure of CPU usage). Figure 1a visualizes the behavior of our training data (21 common HPC benchmarks) with respect to normalized performance counter values. Each data point is the average recorded `POWER_DRAM` and `EXEC` behavior for a training application running in a single resource configuration on our evaluation system. There are 88 unique resource configurations, or possible *labels*, accounting for different combinations of the socket count `S`, whether HyperThreads `HT` are used, and the DVFS frequency (*e.g.,* 2.1 GHz). All 88 data points belonging to a particular application are assigned *the same label*—the resource configuration with the best energy efficiency for that application; *i.e.,* all 88 data points for an application have the same color. With this labeling, a good learner will recognize suboptimal behavior and produce the most energy-efficient settings to use instead. *The problem is clearly complex—no intuitive pattern emerges that obviously maps CPU and memory usage into the most energy-efficient system settings.*

To successfully map these features into accurate predictions, the learner must be able to handle this complexity, which not all learning mechanisms can. Consider Figure 1b, which illustrates the accuracy of a Support Vector Machine (SVM) classifier using a linear kernel. The shaded regions indicate the label (system settings) that the SVM classifier predicts for a range of feature values; the training data is overlaid for comparison. This linear SVM's *recall*—the fraction of system settings that are accurately predicted when simply replaying the training data[1]—is only 45.6%, a clear indication that this classifier is not effective. In contrast, Figure 1c demonstrates a SVM with a radial basis function (RBF) kernel, which achieves 71.0% recall—better, though perhaps still with room for further improvement.

[1]Recall is essentially the lowest barrier to entry; it is simpler than cross-validation.

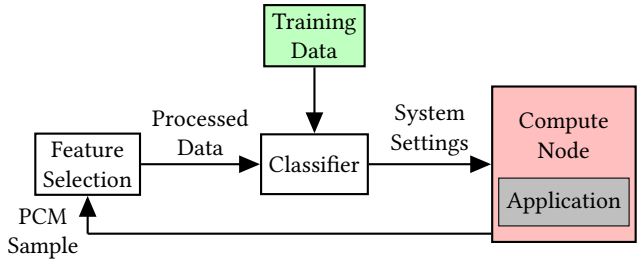


Figure 2: Design for using machine learning classifiers to predict energy-efficient system settings based on performance counter behavior.

3 CLASSIFYING SYSTEM SETTINGS

We propose to predict energy-efficient settings at runtime without the overhead of estimating the behavior of all possible system settings before producing a result. As with many prior works in managing power/energy in HPC systems, we use hardware performance counters to measure application and system behavior.

Figure 2 demonstrates our proposed approach. While an application runs on the compute node, hardware performance counters are polled in the background at regular intervals. For our experiments, we use the PCM tool to collect performance counter data [38]. The data is scaled, then processed using Principal Component Analysis (PCA) to identify which fields correlate well with energy efficiency. We also use feature selection to limit the number of hardware counters used by the classifier to reduce runtime overhead (evaluated in Section 5.2). The classifier predicts the most energy-efficient settings to use, which are then actuated on the system. The process then repeats at the next interval.

3.1 Training Data

A classifier must be trained before it can be used. To collect training data, we characterize the behavior of benchmark applications on the target platform by running them in all possible settings and collect hardware performance counter results. In other words, if there are N different allowable settings, each application is executed N times, or once in each setting. For M training applications, there are $N \times M$ feature vectors in the training set.

Characterization can be time-consuming, but only needs to be done once for a platform and can be completed in a reasonable period of time by keeping application execution times short. Choosing applications that are representative of those that will be used on the system improves the likelihood that the classifier can accurately predict settings during runtime. Additionally, training applications should be chosen that exercise the system hardware components in different patterns to cover a wide range of possible use cases.

Application energy efficiency (EE) is defined as the amount of work completed per unit of energy (J) used. In general, and in our evaluation, a *complete application execution* is the measure of

Table 1: Overview of system-level performance counters.

Performance Counter	Description
EXEC	Instructions per nominal CPU cycle
IPC	Instructions per cycle
FREQ	Frequency relative to nominal CPU frequency
AFREQ	FREQ, excluding the time when the CPU is sleeping
L3MISS	L3 cache line misses
L2MISS	L2 cache line misses
L3HIT	L3 Cache hit ratio
L2HIT	L2 Cache hit ratio
L3MPI	L3 Cache misses per instruction
L2MPI	L2 Cache misses per instruction
READ	Memory read traffic
WRITE	Memory write traffic
INST	Number of instructions retired
Proc Energy	Energy consumed by the processor
DRAM Energy	Energy consumed by the DRAM

completed work. Our classifiers, like many prior works that use performance counters, need a measure to quantify application progress *during* runtime. Low-level hardware performance counters do not have a metric for quantifying true application progress (work completed), but prior works have successfully used instructions retired by the system (INST). Some prior works have even attempted to measure only those instructions considered useful in measuring application progress, *e.g.,* ignoring spinlocks or parallelization/synchronization instructions [9]. Using INST is an imperfect solution for optimizing total application energy efficiency, but suffices, as the evaluation will demonstrate. The classifier then uses the following formulation *as a proxy* to quantify energy efficiency for training:

$$EE = \frac{INST}{J} \tag{1}$$

For each of M applications, the N feature vectors are labeled using that application's most energy-efficient setting. Thus the classifier learns both efficient and inefficient behavior so that it produces an efficient prediction when it observes similar runtime behavior:

$$\textbf{FeatVec}_{mn} \mapsto \underset{i \in N}{\operatorname{argmax}} EE_i \quad \forall m \in M,\ \forall n \in N \tag{2}$$

The total instruction count is not fixed for an application execution—the count typically increases with the application execution time, *e.g.,* due to background processes like PCM or kernel tasks. As such, it is important to note that labeling the most energy-efficient configuration using Eqn. 2 is different than using the minimum-energy execution from the characterization. There are two reasons for using instruction count in computing energy efficiency. First, energy efficiency can be quantified at any point during an execution, making it a useful metric for runtime behavior analysis. Second and more importantly, Eqn. 1 is a function of events happening *at the time.* Total energy requires knowing all events that *will* happen during the application execution, introducing the possibility that the classifiers might be learning something about application input rather than the way hardware events correspond to energy consumption. Accounting for instructions helps to avoid this pitfall.

3.2 Performance Counters

Performance counter metrics are available at different levels of granularity—system, socket, and core. For simplicity, we limit ourselves to system-wide data. Table 1 lists the performance counters that we process for our experiments [51].

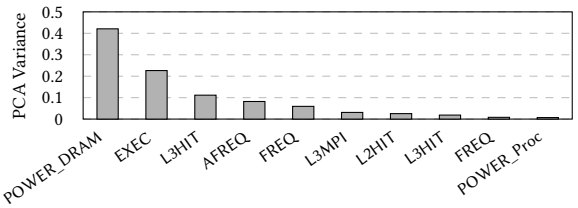


Figure 3: PCA Explained Variance Ratio for the top 10 performance counters, accounting for 99% of variance.

Performance counters are also translated into rates (as needed), which is necessary in order to vary the sampling interval without scaling values (evaluated in Section 5.2). Because we use PCM to collect performance counter metrics, we read more hardware counters than we process (Table 1). In practice, a fielded solution would reduce overhead by only reading and processing performance counters that are used. Most prior works aggressively limit the hardware counters they access, both to reduce sampling overhead and to reduce computation in their models [3, 7, 31].

4 EXPERIMENTAL DESIGN

This section describes our experimental setup, including the evaluation system, applications used for training and evaluation, and the classification algorithms tested. We perform our evaluation on a quad-socket, 80-physical core system with 512 GB DRAM running Ubuntu Linux 14.04 LTS with kernel 4.4.0. With HyperThreads, there are 160 compute threads available, *i.e.,* 20 physical and 20 virtual on each socket.

4.1 Training Applications

Training applications are representative of HPC workloads and are selected from the NAS Parallel Benchmarks [4], Lawrence Livermore Lab's Co-design benchmarks (AMG [16], Kripke [27], LULESH [23], Quicksilver [29]), and Argonne's CESAR Proxy-apps (XSBench [48], RSBench [47]). Other applications include CoMD [28], Berkeley's HPGMG-FV [2], a partial differential equation solver (jacobi), and STREAM [33]. Additionally, we include a characterization of system idling behavior. In total, there are 21 unique applications/characterizations used for training. Each application is configured to run with 160 threads to match the number of compute cores on the evaluation system and to use NUMA memory interleaving.

Each performance counter is used as a *feature* for classification. Performance counter values are converted to rates, normalized, then PCA is applied. Figure 3 quantifies the percentage of variance contributed by the top 10 performance counters in the feature space for our system and training applications, accounting for 99% of the total variance.

4.2 Evaluation Applications

We evaluate classifier performance on four complex bioinformatic HPC applications: HipMer [14], IDBA [41], Megahit [30] and metaSPAdes [37]. These four are the leading applications that perform *de novo* genome assembly, which is one of the most computationally challenging bioinformatics problems. The datasets can be very large (for example, metagenomes can have raw sequence datasets on the order of terabytes), and the algorithms are hard to scale

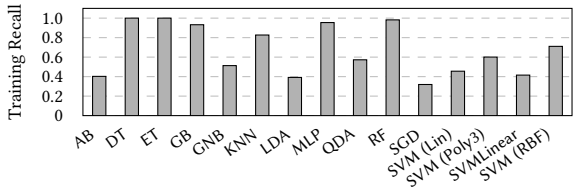


Figure 4: Training data recall for 15 classification algorithm implementations.

efficiently—only HipMer scales efficiently to distributed memory systems. Consequently, *de novo* assemblers are typically run on very large shared-memory systems, with at least 0.5 TB of memory. Thus our experimental platform is typical of the sort of hardware used for this class of applications. A single execution assembling a large genome could take days on our large evaluation system, making it prohibitive to exhaustively characterize the full range of allowable settings, motivating the need for a learning-based solution.

These four applications can also be considered representative of a wider range of HPC applications. They implement complex pipelines, with multiple different stages that require different resource adaptations to run energy-efficiently—some are compute-intensive, some I/O-intensive, and some communication-intensive. Exactly which stages are used and how much they contribute to the performance depends to a large degree on the program configuration and the input datasets. Although they are solving the same problem, they are implemented in very different ways, with different programming languages, different algorithms and different data flows. For example, HipMer can have up to 20 different stages, whereas Megahit may have only a few. Overall, these applications provide a broad coverage of a range of different bioinformatics approaches (frequency counting, graph traversal, alignment, sorting, etc.). The applications are thus prime candidates for our approach of using classification to predict the appropriate settings at runtime.

Like with training applications, we configure each evaluation application to run with 160 threads, except for metaSPAdes which uses 80 threads. We fix the application inputs and configurations for our evaluation, although they support a variety of configurations and inputs which affect performance and energy consumption behavior. Solely to use as a baseline in our evaluation, we perform a time-consuming DVFS-only characterization by running them in each DVFS setting with all 160 virtual cores allocated (for metaSPAdes, all 80 physical cores are used as the baseline instead). From these results we derive a DVFS *Oracle* which knows, for each application, the most energy-efficient static DVFS frequency—*each application has a different most energy-efficient static setting*. Note also that this data is *not* used in classifier training.

4.3 Classification Algorithms

This section identifies and briefly describes the classifiers we use. For data processing and classifier implementations, we use the Machine Learning toolkit scikit-learn, version 0.19.1 [40]. We do not attempt to optimize algorithm performance or prediction accuracy by tuning any implementation knobs. We demonstrate the feasibility of using classification without the need for fine-tuning.

With the labeled training data in Figure 1a (Section 2.2), it is clear that the a useful classification algorithm must handle a complex space. We validate this hypothesis by performing an offline analysis of 15 algorithms using our training data. One of the metrics we looked at was training data recall, which we present in Figure 4. The algorithms are: AdaBoost (AB), Decision Tree (DT), Extra Trees (ET), Gradient Boosting (GB), Gaussian Naive Bayes (GNB), K-Nearest Neighbors (KNN), Linear Discriminant Analysis (LDA), Multi-layer Perceptron (MLP), Quadratic Discriminant Analysis (QDA), Random Forest (RF), Stochastic Gradient Descent (SGD), Support Vector Machine with a linear kernel (SVM (Lin)), SVM with a degree=3 polynomial kernel (SVM (Poly3)), SVM with a different linear kernel (SVMLinear), and SVM with a radial basis function kernel (SVM (RBF)). Some approaches have poor recall and thus are not likely to perform well in practice.

When a mis-prediction occurs, the impact on energy consumption varies—some are suboptimal but still reduce energy consumption over the naive *race* setting, while others actually make it worse. We tested SVM (Lin) (Figure 1b) online—in most cases it reduced energy, but consumed 22% *more energy* than *race* with the Megahit application, demonstrating the importance of selecting a good classifier. We choose five promising algorithms that support a range of different classification techniques to use in the evaluation:

(1) ET – an extremely randomized decision tree, similar to Random Forest [15].
(2) GB – fits multiple regression trees on the negative gradient of the deviance loss function [13, 40].
(3) KNN – a simple majority vote of the nearest neighbors from the training data ($k = 5$, by default).
(4) MLP – a neural network optimizing the log-loss function using *lbfgs*, a *tanh* activation function, and four layers [17].
(5) SVM – a maximum margin classifier using a radial basis function (RBF) kernel.

The only classifier with non-default configurations is MLP, in order to add additional layers to better represent deep learners, and to specify the activation and solver functions. Our evaluation compares the energy consumption of real application executions when using these five classifiers in different configurations.

5 EVALUATION

We now evaluate the effectiveness of using machine learning classifiers to predict energy-efficient system settings during application runtime. First, we compare against the naive *race* setting, *i.e.,* all sockets allocated with HyperThreads and DVFS set to TurboBoost, and against a DVFS *Oracle*. We then quantify how varying sampling/prediction intervals and the number and types of features impacts classifier effectiveness. We discuss how application dynamics affect the classifiers and evaluate the overhead of different parts of the classifier runtime. Finally, we explore using separate classifiers for taskset and DVFS, then discuss limitations.

5.1 Reducing Energy Consumption

Figure 5 demonstrates how effective runtime classification is at reducing energy consumption, normalized to the *race* setting (dashed line). On average across all four applications, energy consumption is reduced by 19.3%—there is a 31.3% increase in runtime and 38.5%

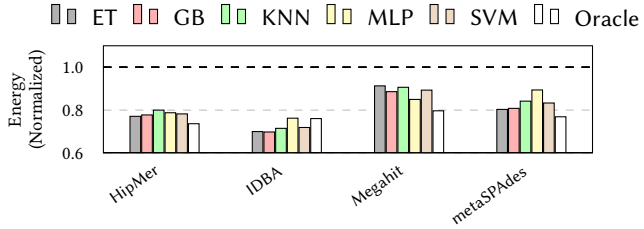


Figure 5: Average energy consumption using 4 performance counters at 5 second prediction intervals (lower is better).

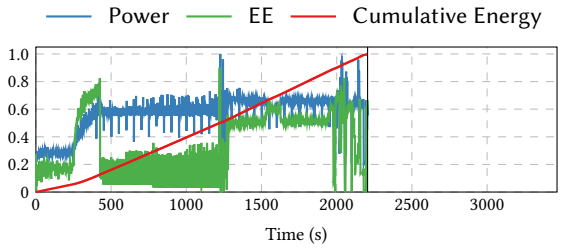
reduction in average power consumption. Extrapolating from these empirical results, a hardware over-provisioned, power-constrained cluster could increase throughput by 24%; the *Oracle* could achieve up to 30% increase in throughput. The most extreme results are from the IDBA and Megahit applications. For IDBA, each classifier outperforms the *Oracle*—the average energy savings is 28.1%, and as much as 30.2% when using the GB classifier. Conversely, for Megahit, the classifiers have the highest energy consumption over the *Oracle*, though still always better than *race*. Megahit saves 11% energy on average—8.7% in the worst case with ET, and 15% at best with MLP. These applications are discussed further in Section 5.3.

Computing the *Oracle* requires expensive offline characterization to determine the best static setting for each application and its input/configuration, making it impractical to determine for most applications and difficult to beat. It also does not have any overhead except for running PCM in the background. The *Oracle* can only be beat when an application does not require its full taskset (socket and HyperThreads) allocation to run efficiently. HPC applications are designed to parallelize well, meaning this is not the typical use case, but opportunities do occur, *e.g.*, during prolonged memory- or I/O-intensive phases.

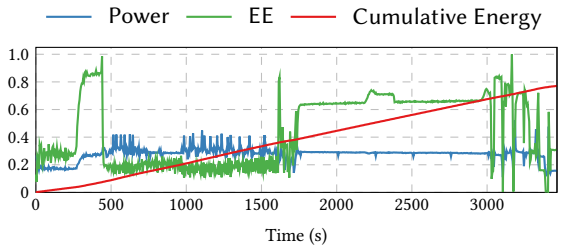
To demonstrate how energy savings are achieved, Figure 6 shows the runtime, power, energy efficiency, and cumulative energy consumption for executions of the HipMer application (Y values are normalized across both time series for each metric). Figure 6a runs HipMer in the *race* setting, and although the runtime is short, the high power also results in poor energy consumption. In contrast, Figure 6b demonstrates running with the ET classifier—the execution is 29% longer, but the 40% average power savings results in nearly 20% less energy consumption in total. Each figure shows clear phases in the execution, indicated by relatively long-term changes in both power and energy efficiency (per Eqn. 1, a function of instruction rate and power). Classifiers adapt to changes in feature values like instruction rate and power by changing their predictions to run the application more efficiently, thus decreasing total energy consumption.

5.2 Classifier Interval and Feature Selection

The sampling and prediction interval dictates how quickly a classifier can respond to changes in application behavior, but also incurs overhead and affects a classifier's susceptibility to noise. Figure 7 shows the normalized energy consumption, averaged across applications, for each classifier at 1, 5, and 10 second intervals. The thin dotted line at 0.75 is the average normalized energy consumption for the *Oracle*. The 1 second interval is clearly the worst, only



(a) HipMer in naive static *race* setting.



(b) HipMer with ET classifier at 5 second intervals.

Figure 6: Execution time, power, energy efficiency, and energy consumption behavior for the HipMer application (a) without and (b) with classification.

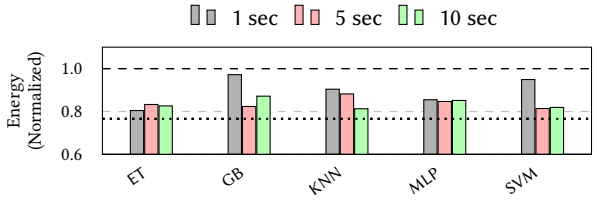
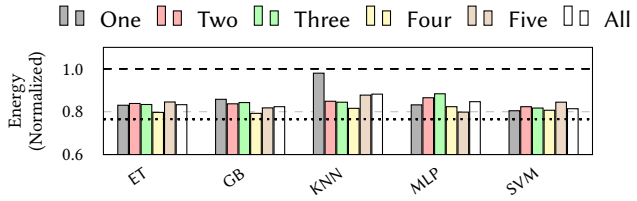


Figure 7: Average energy consumption for different sampling/prediction intervals (lower is better).

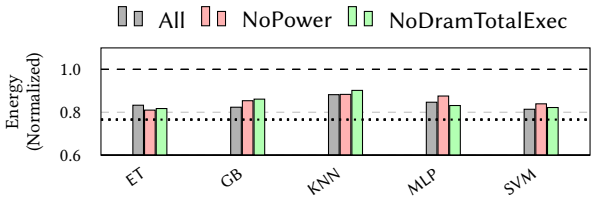
outperforming the others for the ET classifier. It is closer to the *race* setting than the *Oracle* for GB and SVM, but still better than *race*. Both the 5 and 10 second intervals perform well—although the 10 second interval does better for one particular classifier (KNN), the 5 second interval is similar on average and is more responsive. We elect to use the 5 second interval for the remainder of our experiments, as it provides a good tradeoff of energy efficiency and responsiveness to changing application behavior.

In Figure 3 (Section 4.1) we quantified the variation contribution of different features (performance counters). Now we evaluate how varying the features impacts classifier behavior since the number and types of features contribute to runtime complexity and prediction accuracy. We run with 5 second sampling/prediction intervals and present the results in Figure 8.

Figure 8a varies the number of features used. `POWER_DRAM` is the only feature used for *One*, `POWER_DRAM` and `EXEC` are used for *Two*, etc. For *All*, all 16 features are used. Using a single feature works surprisingly well for most classifiers, but performs poorly with KNN, which too often predicts settings that use only a single socket. It is a little surprising that a single feature can be so effective otherwise, but `POWER_DRAM` does account for 42% of the explained variance, and is correlated with other performance counters like cache hit



(a) Average energy consumption based on number of features.



(b) Average energy consumption based on types of features.

Figure 8: Average energy consumption, varying the number and types of available features (lower is better).

and miss rates. Using two or three features also works quite well, but four features is the best for most classifiers, accounting for 84% of the variance. The *Four* configuration was previously broken down by application in Section 5.1.

Figure 8b selectively removes available features. *NoPower* drops the `POWER_DRAM`, `POWER_Proc`, and `POWER_Total` features. This is an important scenario, as not all systems have power/energy counters available at runtime (training data would have to be collected with additional power/energy instrumentation). When all three power-related features are ignored, `L3MISS_Rate` is used as the first feature, and `EXEC` remains the second. (If `POWER_DRAM` is excluded exclusively, `POWER_Total` becomes the primary feature instead.) Offline analysis also shows that `READ_Rate` can be used as the primary feature, depending on the exact training data. *NoDramTotalExec* drops `POWER_DRAM`, `POWER_Total`, and `EXEC`, making `AFREQ` and `WRITE_Rate` the primary and secondary features. In general, *All* outperforms the other two, but *NoPower* does slightly better for the ET classifier, as does *NoDramTotalExec* with MLP. None of the configurations perform poorly, demonstrating that the classification approach is robust to different numbers and types of features.

5.3 Application Dynamics

The results thus far raise the question: why does the *Oracle* often perform better than the classifiers? First, we recall that the *Oracle* requires each application to be characterized across all DVFS settings with their current input and configuration, so is not practical to discover for most applications in practice. Additionally, the *Oracle* does not incur any overhead except sampling PCM (runtime overheads are quantified next in Section 5.4), and can only be beat when applications do not require all sockets and HyperThreads. In fact, for 3 of the 4 evaluation applications, classification usually achieves energy consumption close to that of the *Oracle*, and sometimes better.

Energy consumption penalties are incurred primarily by: (1) overhead from changing settings, and (2) running in inefficient settings. While there is no particular threshold for determining when

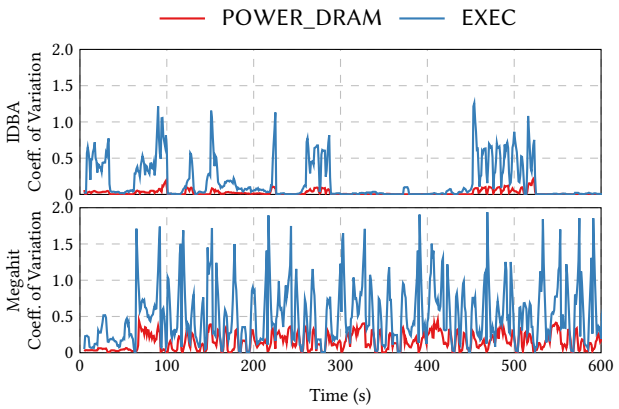


Figure 9: Runtime coefficient of variation over a 5 second sliding window (lower values indicates more stable application behavior).

Table 2: Frequency of settings changes for ET classifier in the *All* configuration.

Application	Prediction	Taskset	DVFS
HipMer	32.7%	25.6%	23.1%
IDBA	39.9%	37.4%	18.8%
Megahit	46.6%	37.9%	39.8%
metaSPAdes	41.9%	32.9%	28.6%

changes to performance counter values result in a new prediction, we quantify the difficulty in controlling applications by examining performance counters' coefficients of variation during the course of an application execution. Since applications move through phases, a high coefficient of variation for an execution does not cause problems on its own. The difficulty arises when performance counter values fluctuate rapidly enough to cause the classifiers to predict new system settings, constantly incurring actuation overhead and spending time in suboptimal settings.

In Figure 9, we examine `POWER_DRAM` and `EXEC` performance counter behaviors without any runtime actuation, with all sockets/cores allocated and using the maximum DVFS frequency *without* TurboBoost (to avoid Turbo-related fluctuations). For the first 10 minutes of each execution, we compute coefficient of variation over 5 second windows to demonstrate the variability that our classifiers see. We present results for IDBA, which in most cases outperforms the *Oracle*, and for Megahit, which exhibits rapid fluctuations and consistently performs worse than the *Oracle* (but still better than *race*). While IDBA does exhibit fluctuations, they are typically short-lived—most of the time the performance counter values are relatively stable. In contrast, Megahit is constantly noisy, resulting in frequent changes to system settings.

For example, Table 2 quantifies how frequently predictions and system settings change when using the ET classifier in the *All* configuration. Megahit has the highest frequency of changing predictions, but more importantly (and difficult to quantify), is how extreme the fluctuations' impacts are. With IDBA, the classifier prefers to switch between one socket and four sockets, always with HyperThreads enabled; DVFS is typically around 1.8 GHz, other times running in TurboBoost. With Megahit, the classifier

Table 3: Classification runtime overheads.

Stage	Average Overhead (ms)
Init – Scaling/PCA	4.50
Init – ET	29.01
Init – GB	3456.34
Init – KNN	3.50
Init – MLP	1278.30
Init – SVM	68.92
PCM Sampling	14.70
Scaling/PCA	0.08
Predict – ET	0.75
Predict – GB	0.36
Predict – KNN	0.39
Predict – MLP	0.14
Predict – SVM	0.13
Actuation	≈ 100

chooses four sockets with HyperThreads, four sockets without HyperThreads, and one socket with HyperThreads; DVFS is consistently around 1.7 or 1.8 GHz, periodically running in TurboBoost when all cores are being used. Megahit also poses an additional technical challenge—it appears to constantly destroy and spawn threads, which sometimes makes managing its taskset difficult.

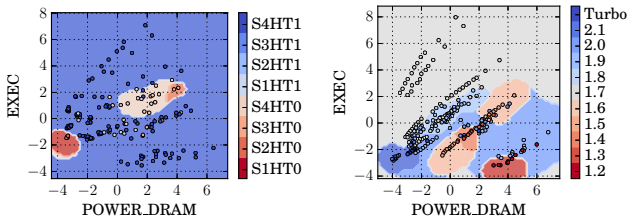
In short, the dynamic behavior that applications exhibit can make predicting energy-efficient settings at runtime a challenge. Handling variability is not something the classifiers learn during training. However, any resource management system that is dependent on runtime feedback must find a good balance between responsiveness and susceptibility to noisy data—a challenge not limited to using classification.

5.4 Overhead

For classification to be practical, the runtime overhead must be reasonable. Table 3 breaks down the runtime overhead on our evaluation system. First, the data transformation (Scaling/PCA) and classifier must be initialized. Initialization incurs the highest overhead, but only needs to be performed once and is trivial compared to total application runtime. All classifiers we evaluated initialize within a few seconds, and some are much faster, initializing within a few dozen milliseconds.

To estimate the overhead incurred by reading the performance counters (PCM Sampling), we configure PCM to poll counters 1000 times at an interval that is faster than achievable (so it does not wait between reads), and time the execution. As a result, the average value for PCM also accounts for its initialization and teardown time.

Transforming the PCM data (Scaling/PCA) prior to running the classifier is extremely fast, averaging about 80 μs. Prediction overheads vary by classifier, but the average overheads for each are strictly less than 1 ms. The actuation overhead, while high compared to sampling and prediction, is a task that any runtime resource allocator that manages DVFS and taskset has to incur. Although dependent on application behavior, in 1/2 to 3/4 of cases we find that no actuation overhead is incurred since the classifier's prediction does not change at each sampling interval (Table 2). When there is a change, the overhead is on the order of 100 ms, which is consistent with observations on other server-class systems [21]. It is also import to note that PCM, the classifier, and the actuator



(a) Taskset only, recall=0.869. **(b) DVFS only, recall=0.645.**

Figure 10: Training data and learned decision boundaries for SVM when classifying for taskset and DVFS separately.

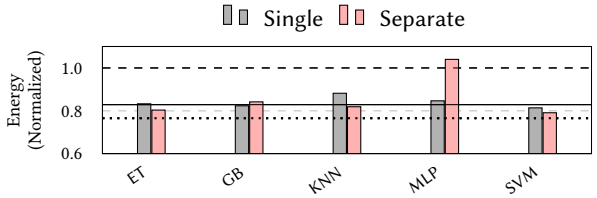


Figure 11: Average energy consumption when using a single or separate classifiers for system knobs (lower is better).

run in the background, so the application does not stop making progress while the resource management components work.

As mentioned previously, we have not attempted to optimize any of these overhead results. We expect that it is readily possible to reduce overhead, particularly in reading performance counters (PCM samples more performance counters than we actually use) and integrating with other tools for managing taskset [45]. Additionally, if initialization time is a concern, classifier state could be stored and reloaded between executions, or the classifier could just run continuously on the system.

5.5 Using Separate Classifiers

The evaluation thus far has used a single classifier to predict a label that is a combination of both taskset (socket allocation, including HyperThreads) and the DVFS frequency (including TurboBoost). With 8 taskset options and 11 DVFS settings, there are 88 total possible labels for the classifier to choose from. After labeling the training data, only a subset of these 88 are used, but the size of that subset would likely increase with additional training data.

An alternative approach is for one classifier to predict the taskset and a separate classifier to predict the DVFS frequency. Figure 10 visualizes the training and recall for SVM (like Figure 1c in Section 2) using separate classifiers. Note that there are fewer data points, as DVFS is fixed at TurboBoost when training for taskset, and taskset is fixed at all sockets and HyperThreads when training for DVFS. Taskset's recall is quite good while DVFS is not as good as before. There is, of course, additional overhead in running two classifiers instead of one, but Section 5.4 demonstrated that data processing overhead is not significant (and there is still only a single instance of PCM and the actuators running).

Figure 11 quantifies the behavior when we use two separate classifiers simultaneously. In each case, the same type of classifier is used for both taskset and DVFS prediction. In 3 of 5 cases, *Separate* actually outperforms the *Single* approach. However, it is

also possible for *Separate* to perform rather poorly, as seen with the MLP classifier. The average MLP behavior is representative of the applications tested (*i.e.,* not caused by an outlier)—in three of the four applications evaluated, MLP classifying taskset and DVFS separately performed worse than the *race* setting. Of course, taskset and DVFS prediction may benefit from using different classification algorithms. We tested taskset-only and DVFS-only classifiers in isolation and empirically determined that, for our system and applications, the ET classifier works best for taskset prediction and the SVM classifier works best for DVFS prediction. The solid horizontal line indicates the energy consumption for this ET/SVM classifier mix. While this avoided the poor results seen with MLP, it actually performed slightly worse than 3 of the 4 remaining *Single* classifier approaches. Using a single, unified classifier that learns both taskset and DVFS together ultimately produces more reliable predictions.

5.6 Discussion of Results and Limitations

Our evaluation demonstrates that machine learning classification driven by low-level features is an effective approach for improving energy efficiency. In fact, a variety of different classifiers are useful for predicting energy-efficient system settings at runtime, even without classifier tuning. No single classifier appears to have a clear advantage over others, though future work on fine-tuning the training data and classifiers, combined with evaluations on other platforms, may eventually produce a near-optimal implementation.

We also did not attempt to optimize the runtime's overhead—reading performance counters, data transformations, or system actuation. Although low, there is still room for improvement, which could support faster sampling and prediction intervals and further improve energy consumption.

Because applications are launched with a fixed number of threads, those threads are forced to share compute cores when we bind applications to fewer sockets or disable HyperThreads during runtime. This over-subscription is often not as efficient as matching the thread count to the available cores, *e.g.,* running 80 threads on our 80 physical cores is usually more efficient than binding 160 threads to those 80 cores. Integration with application-level parallelism can further improve energy efficiency and reduce resource contention when binding applications to smaller tasksets [45].

On large systems such as our evaluation platform, the processor and DRAM consume the majority of system power, which we are able to measure [8, 38]. There are other components like hard disks and network interfaces that are not currently accounted for. Similarly, extending the approach to support heterogeneous architectures like those using GPUs would also be beneficial. While it is not common practice to instrument these other components for power/energy monitoring, such feedback could help resource management solutions achieve better total energy efficiency.

6 RELATED WORK

Power and energy in HPC systems is a growing concern, though prior work in the area has often not allowed trading performance for power or energy savings. For example, Adagio uses DVFS to save energy with less than 1% increase in runtime [42]. Other work depends on accurate prediction of a code's critical path to reduce power where it will not slow down an application [22, 32]. Patki et al. propose to better utilize available power with hardware overprovisioning to increase total system throughput [39]. Sarood et al. have shown similar results: hardware over-provisioning increases performance given a power cap [43]. Hardware over-provisioning acknowledges that compute resources are no longer the primary factor limiting cluster size—power is, allowing us to more aggressively trade performance and power/energy consumption.

Other works demonstrate that hardware performance counters can drive solutions for modeling and improving power/energy consumption [31, 44, 49, 52]. Using Dynamic Concurrency Throttling and DVFS to reduce energy consumption without performance loss, Curtis-Maury et al. use hardware events to create an energy-aware logistic regression model to predict performance and power [7].

As the number of system settings increases and their interaction becomes more complicated, several approaches have turned to machine learning to manage them. Paragon [9] and Quasar [10] guarantee quality-of-service constraints in heterogeneous data centers using a scheduler based on collaborative filtering. LEO uses a hierarchical Bayesian model to minimize energy for different system utilization requirements [34, 35]. These approaches all estimate the performance and power of every possible system configuration, then search those estimates to find the best configuration that meets their operating constraints. The need to estimate every configuration's behavior is expensive, requiring half a second [35] to several seconds [9] of overhead. In contrast, the approach in this paper simply returns the best system settings without predicting their actual energy efficiency, an approach that requires orders of magnitude less overhead (see Table 3 in Section 5.4). Ferroni et al. use classification based on hardware events to select the best power model for an application from a predetermined set, and then assign resources to that application in a multi-tenant virtualized infrastructure [12]. This approach and the proposed approach are complementary, in that Ferroni et al. assign nodes as resources, while we fine-tune resource usage within a node.

Our proposed approach is most closely related to other node-level approaches for managing performance and energy. PUPiL maximizes node performance given a power cap by adjusting system settings to the particular needs of an application [53]. Chasapis et al. maximize performance for power-capped NUMA nodes by recognizing the effect that manufacturing variability can have on individual core performance [6]. Both of these approaches maximize performance for a given power constraint. Neither, however, is capable of minimizing energy, which requires changing both power and performance as in our proposed approach. ParallelismDial is a node-level approach for managing application-level parallelism to increase energy efficiency [45]. ParallelismDial has similar goals to our proposed approach, but they are complementary. It works at the application level, while our approach operates at the system level. In future work, it would likely be beneficial to combine the two to further reduce overhead and improve energy savings. This combination has proved fruitful for embedded systems [19, 20, 25, 50] and will likely benefit HPC as well.

7 CONCLUSION

We argue that reducing application energy consumption reduces the cost (in Joules) of scientific insight for computing systems, and

even increases the throughput of power-constrained systems. Optimizing for energy efficiency is hard, however, especially without modifying applications—it requires an approach that can map low-level features into energy-efficient system settings. We explore 15 different classification schemes to perform this complex mapping and find that some are much better suited to this problem than others. Having identified suitable learning algorithms, we demonstrate five that dramatically reduce energy consumption for production HPC genome assembly applications with low overhead.

ACKNOWLEDGMENTS

This research is supported by the NSF (CCF-1439156, CNS-1526304), the DARPA BRASS program, and a DOE Early Career award. This work is supported in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research.

REFERENCES

[1] Bilge Acun, Phil Miller, and Laxmikant V. Kale. 2016. Variation Among Processors Under Turbo Boost in HPC Systems. In *ICS*.
[2] Mark F. Adams, Jed Brown, John Shalf, Brian van Straalen, Erich Strohmaier, and Sam Williams. 2014. *HPGMG 1.0: A Benchmark for Ranking High Performance Computing Systems.* Technical Report LBNL-6630E. LBNL.
[3] Claudia Alvarado, Dan Tamir, and Apan Qasem. 2015. Realizing Energy-efficient Thread Affinity Configurations with Supervised Learning. In *IGSC*.
[4] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. 1991. The NAS Parallel Benchmarks–Summary and Preliminary Results. In *SC*.
[5] Michael Berry, Thomas E. Potok, Prasanna Balaprakash, Henry Hoffmann, Raju Vatsavai, Prabhat, and Robinson Pino. 2015. Machine Learning and Understanding for Intelligent Extreme Scale Scientific Computing and Discovery. (2015).
[6] Dimitrios Chasapis, Marc Casas, Miquel Moretó, Martin Schulz, Eduard Ayguadé, Jesus Labarta, and Mateo Valero. 2016. Runtime-Guided Mitigation of Manufacturing Variability in Power-Constrained Multi-Socket NUMA Nodes. In *ICS*.
[7] Matthew Curtis-Maury, Ankur Shah, Filip Blagojevic, Dimitrios S. Nikolopoulos, Bronis R. de Supinski, and Martin Schulz. 2008. Prediction Models for Multi-dimensional Power-performance Optimization on Many Cores. In *PACT*.
[8] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. 2010. RAPL: Memory Power Estimation and Capping. In *ISLPED*.
[9] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware Scheduling for Heterogeneous Datacenters. In *ASPLOS*.
[10] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and QoS-aware Cluster Management. In *ASPLOS*.
[11] Sudip S Dosanjh, Shane Canon, Jack Deslippe, Kjiersten Fagnan, Richard A Gerber, Lisa Gerhardt, Jason Hick, Douglas Jacobsen, David Skinner, and Nicholas J Wright. 2013. Extreme Data Science at the National Energy Research Scientific Computing (NERSC) Center.. In *PARCO*.
[12] Matteo Ferroni, Andrea Corna, Andrea Damiani, Rolando Brondolin, Juan A. Colmenares, Steven Hofmeyr, John D. Kubiatowicz, and Marco D. Santambrogio. 2017. Power Consumption Models for Multi-Tenant Server Infrastructures. *ACM TACO* 14, 4, Article 38 (Nov. 2017).
[13] Jerome H. Friedman. 2001. Greedy Function Approximation: A Gradient Boosting Machine. *Ann. Statist.* 29, 5 (Oct. 2001).
[14] Evangelos Georganas, Aydın Buluç, Jarrod Chapman, Steven Hofmeyr, Chaitanya Aluru, Rob Egan, Leonid Oliker, Daniel Rokhsar, and Katherine Yelick. 2015. HipMer: An Extreme-Scale De Novo Genome Assembler. In *SC*.
[15] Pierre Geurts, Damien Ernst, and Louis Wehenkel. 2006. Extremely Randomized Trees. *Machine Learning* 63, 1 (01 Apr 2006).
[16] Van Emden Henson and Ulrike Meier Yang. 2002. BoomerAMG: A Parallel Algebraic Multigrid Solver and Preconditioner. *Appl. Numer. Math.* (April 2002).
[17] Geoffrey E. Hinton. 1989. Connectionist Learning Procedures. *Artificial Intelligence* 40, 1 (1989).
[18] Henry Hoffmann. 2013. Racing and Pacing to Idle: An Evaluation of Heuristics for Energy-aware Resource Allocation. In *HotPower*.
[19] Henry Hoffmann. 2014. CoAdapt: Predictable Behavior for Accuracy-Aware Applications Running on Power-Aware Systems. In *ECRTS*.
[20] Henry Hoffmann. 2015. JouleGuard: Energy Guarantees for Approximate Applications. In *SOSP*.
[21] Connor Imes, David H. K. Kim, Martina Maggio, and Henry Hoffmann. 2016. Portable Multicore Resource Management for Applications with Performance Constraints. In *MCSoC*.
[22] Nandini Kappiah, Vincent W. Freeh, and David K. Lowenthal. 2005. Just In Time Dynamic Voltage Scaling: Exploiting Inter-Node Slack to Save Energy in MPI Programs. In *SC*.
[23] Ian Karlin, Jeff Keasler, and Rob Neely. 2013. *LULESH 2.0 Updates and Changes.* Technical Report LLNL-TR-641973.
[24] David H. K. Kim, Connor Imes, and Henry Hoffmann. 2015. Racing and Pacing to Idle: Theoretical and Empirical Analysis of Energy Optimization Heuristics. In *CPSNA*.
[25] Minyoung Kim, Mark-Oliver Stehr, Carolyn Talcott, Nikil Dutt, and Nalini Venkatasubramanian. 2013. xTune: A Formal Methodology for Cross-layer Tuning of Mobile Embedded Systems. *ACM TECS* 11, 4, Article 73 (2013).
[26] Peter Kogge, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Stephen Keckler, Dean Klein, and Robert Lucas. 2008. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. (28 Sept. 2008).
[27] Adam J. Kunen, Teresa S. Bailey, and Peter N. Brown. 2015. KRIPKE - A Massively Parallel Transport Mini-App. In *American Nuclear Society M&C*. http://www.osti.gov/scitech/servlets/purl/1229802
[28] Los Alamos National Laboratory. 2016. CoMD. (2016). https://github.com/ECP-copa/CoMD
[29] Lawrence Livermore National Laboratory. 2017. Co-design at Lawrence Livermore National Lab – Quicksilver. (2017). https://codesign.llnl.gov/quicksilver.php
[30] Dinghua Li, Chi-Man Liu, Ruibang Luo, Kunihiko Sadakane, and Tak-Wah Lam. 2015. MEGAHIT: an ultra-fast single-node solution for large and complex metagenomics assembly via succinct de Bruijn graph. *Bioinformatics* 31, 10 (2015).
[31] Simone Libutti, Giuseppe Massari, Patrick Bellasi, and William Fornaciari. 2014. Exploiting Performance Counters for Energy Efficient Co-Scheduling of Mixed Workloads on Multi-Core Platforms. In *PARMA-DITAM*.
[32] Aniruddha Marathe, Peter E. Bailey, David K. Lowenthal, Barry Rountree, Martin Schulz, and Bronis R. de Supinski. 2015. A Run-Time System for Power-Constrained HPC Applications. In *ISC*.
[33] John D. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE TCCA Newsletter* (Dec. 1995).
[34] Nikita Mishra, Connor Imes, John D. Lafferty, and Henry Hoffmann. 2018. CALOREE: Learning Control for Predictable Latency and Low Energy. In *ASPLOS*.
[35] Nikita Mishra, Huazhe Zhang, John D. Lafferty, and Henry Hoffmann. 2015. A Probabilistic Graphical Model-based Approach for Minimizing Energy Under Performance Constraints. In *ASPLOS*.
[36] NERSC. 2017 (accessed Jan, 2018). GENEPOOL. (2017 (accessed Jan, 2018)). http://www.nersc.gov/users/computational-systems/genepool/
[37] Sergey Nurk, Dmitry Meleshko, Anton Korobeynikov, and Pavel Pevzner. 2016. metaSPAdes: a new versatile de novo metagenomics assembler. *ArXiv e-prints* (April 2016). arXiv:q-bio.GN/1604.03071
[38] opcm. 2016. Processor Counter Monitor (PCM). (2016). https://github.com/opcm/pcm
[39] Tapasya Patki, David K. Lowenthal, Anjana Sasidharan, Matthias Maiterth, Barry L. Rountree, Martin Schulz, and Bronis R. de Supinski. 2015. Practical Resource Management in Power-Constrained, High Performance Computing. In *HPDC*.
[40] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (Nov. 2011). http://dl.acm.org/citation.cfm?id=1953048.2078195
[41] Yu Peng, Henry CM Leung, Siu-Ming Yiu, and Francis YL Chin. 2012. IDBA-UD: a de novo assembler for single-cell and metagenomic sequencing data with highly uneven depth. *Bioinformatics* 28, 11 (2012).
[42] Barry Rountree, David K. Lownenthal, Bronis R. de Supinski, Martin Schulz, Vincent W. Freeh, and Tyler Bletsch. 2009. Adagio: Making DVS Practical for Complex HPC Applications. In *ICS*.
[43] Osman Sarood, Akhil Langer, Laxmikant Kale, Barry Rountree, and Bronis de Supinski. 2013. Optimizing Power Allocation to CPU and Memory Subsystems in Overprovisioned HPC Systems. In *CLUSTER*.
[44] Hiroshi Sasaki, Yoshimichi Ikeda, Masaaki Kondo, and Hiroshi Nakamura. 2007. An Intra-task Dvfs Technique Based on Statistical Analysis of Hardware Events. In *CF*.
[45] Srinath Sridharan, Gagan Gupta, and Gurindar S. Sohi. 2013. Holistic Run-time Parallelism Management for Time and Energy Efficiency. In *ICS*.
[46] ExaOSR Team. 2012. Key Challenges for Exascale OS/R. (15 June 2012). https://collab.mcs.anl.gov/display/exaosr/Challenges
[47] John R. Tramm, Andrew R. Siegel, Benoit Forget, and Colin Josey. 2014. Performance Analysis of a Reduced Data Movement Algorithm for Neutron Cross Section Data in Monte Carlo Simulations. In *EASC*.
[48] John R. Tramm, Andrew R. Siegel, Tanzima Islam, and Martin Schulz. 2014. XSBench – The Development and Verification of a Performance Abstraction for

Monte Carlo Reactor Analysis. In *PHYSOR*.

[49] Ghislain Landry Tsafack Chetsa, Laurent Lefèvre, Jean-Marc Pierson, Patricia Stolf, and Georges Da Costa. 2013. Exploiting Performance Counters to Predict and Improve Energy Performance of HPC Systems. *Future Generation Computer Systems* (Aug. 2013). https://hal.inria.fr/hal-00925306

[50] Vibhore Vardhan, Wanghong Yuan, Albert F. Harris III, Sarita V. Adve, Robin Kravets, Klara Nahrstedt, Daniel Grobe Sachs, and Douglas L. Jones. 2009. GRACE-2: Integrating Fine-Grained Application Adaptation with Global Adaptation for Saving Energy. *IJES* 4, 2 (2009).

[51] Thomas Willhalm. 2014. Intel PCM Column Names Decoder Ring. (18 July 2014). https://software.intel.com/en-us/blogs/2014/07/18/intel-pcm-column-names-decoder-ring

[52] Xingfu Wu, Valerie Taylor, Jeanine Cook, and Philip J. Mucci. 2016. Using Performance-Power Modeling to Improve Energy Efficiency of HPC Applications. *Computer* 49, 10 (Oct. 2016).

[53] Huazhe Zhang and Henry Hoffmann. 2016. Maximizing Performance Under a Power Cap: A Comparison of Hardware, Software, and Hybrid Techniques. In *ASPLOS*.