Phase Recognition from Power Traces of HPC Workloads

Joseph Granados, Jake Probst, Nick Armour, Jeff Bahns, Suzanne Rivoire
Department of Computer Science
Sonoma State University
Rohnert Park, CA, USA

Chung-Hsing Hsu
Oak Ridge National Laboratory
Oak Ridge, TN, USA

***Abstract*—Prior work has shown that power consumption traces of HPC workloads exhibit distinctive statistical characteristics, which allows the workload that generated a given power trace to be inferred with high accuracy. However, these power signatures apply to the entire power trace, with no ability to break it down further into phases or to recognize novel combinations of known workloads.**

In this work, we propose and evaluate a mechanism for partitioning a power trace into phases and matching each phase to a known kernel or workload. We evaluate this technique on a set of 388 power traces collected from 21 benchmarks, including CPU-intensive system stressors; the NAS Parallel Benchmarks; and Mahout data analytics workloads. Our technique is able to, on average, attribute 78% of the points in a concatenated trace to the correct kernel.

I. Introduction

Power constraints and energy costs loom large in high-performance computing. Understanding workloads' power consumption behavior can help schedulers to optimize performance under a power constraint. Prior work has shown that HPC workloads exhibit distinctive power consumption patterns and that, given a power consumption trace, the workload that generated it can be predicted with high accuracy [1]. However, this past work was limited in two related respects:

1) The workload identification operates at the granularity of an entire power trace or application, with no ability to drill down into parts of the trace.
2) In order to match a power trace to the workload that generated it, there must be prior power traces collected from that exact workload. Novel combinations of known kernels are unrecognizable.

In this work, we propose a method of examining a power trace and identifying the series of known kernels that it comprises, rather than matching the entire trace to a single application. In addition to providing finer-grained information about the trace than previous approaches, this ability also simplifies collection of the training database of power traces, since it only needs to contain examples of each individual kernel rather than every possible combination of kernels.

In our experiments, this phase detector correctly attributes an average of 78% of the points in a power trace to the correct kernel that generated them. The accuracy of the underlying whole-trace workload identifier is 85%, which suggests that our approach is highly successful at correctly partitioning the trace into phases.

After covering related work in Section II, we describe our 4-step process for segmenting a trace into its constituent kernels in Section III. In Section IV, we present our experimental setup and results. We conclude with a discussion of future work to extend and generalize our techniques.

II. Related Work

Many previous studies have proposed methods of identifying phases in a program's execution. Some of the techniques were originally developed in the context of reconfigurable computing, with the goal of deciding when the workload had changed sufficiently to justify reconfiguring the processor [2]. More recently, phase recognition has become important in selecting appropriate dynamic voltage and frequency scaling (DVFS) settings for processors.

Most of this work, even if the goal is power and energy savings, recognizes phases based on program characteristics or hardware performance counters rather than directly from power traces. A common approach is to create a vector of performance counter values (an execution vector) for each time-slice of a program execution. Then, phase changes can be detected when the distance between adjacent vectors exceeds some threshold, and phase recognition can also be achieved with a distance computation. Chetsa et al. use this technique to partition HPC workloads into CPU-, memory-, and I/O-intensive phases with the goal of saving energy by selecting appropriate hardware settings for each phase [3]. Paul et al. used hardware counters to find phases in heterogeneous CPU-GPU workloads to drive DVFS decisions. Like us, they focus on applications that consist of a series of diverse kernels [4]. Ioannou et al. use patterns of MPI calls as the basis for their program phases [5]. Peraza et al., citing the lack of portability of hardware performance counters, use offline interprocedural trace analysis to determine DVFS phases [6]. One of few studies to use power as the basis for phase detection was done by Isci et al., who used vectors of estimated power values in 22 different parts of a processor chip at each sampled execution

This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

point [7]. However, the actual phase detection and recognition techniques they used are based on vector distance, similar to Chetsa's work with execution vectors.

Because we are working with only a single time series (the power trace) and attempting to map each of its phases to a specific kernel rather than a generic workload class, the techniques we use draw more heavily from time series analysis. Our phase recognition problem is essentially the inverse of the well-known problem of subsequence matching [8]. In subsequence matching, the training set would consist of long traces, and the task is to match a shorter query trace to a subset of a longer trace. For us, the training set consists of short traces or kernels, and our task is to decompose a long trace into a concatenation of these shorter kernels.

III. Phase Detection Method

We assume a training set of labeled power traces: that is, a set of tuples $(\mathrm{PowerTrace}, \mathrm{Workload})$. In this report, we will refer to the workloads represented in this training set as *kernels* because they are the smallest units that our method will be able to recognize, even if they are internally complex.

We also assume an unlabeled test trace, consisting of a concatenation of kernels that are present in the training set. The specific power traces being concatenated are not assumed to appear in the training set, but the workloads to which they belong are.

Our goal is to decompose the test trace into its constituent kernels and accurately identify each kernel. The output of our algorithm will be a list of non-overlapping time intervals, each one labeled with the kernel inferred to be running in each interval.

Our approach to decomposing a trace into phases is as follows:

1) Identify *change points* corresponding to possible phase boundaries in the test trace.
2) Identify candidate phases (possibly overlapping), where each phase is an interval starting at either the beginning of the trace or a change point, and ending at either a change point or the end of the trace.
3) Attempt to identify the task type of each candidate phase.
4) Choose a final partition of the trace from among the candidate phases.

A. Change Point Detection

Detecting abrupt changes in the statistical properties of a time series is a well studied problem with applications in a variety of fields. Fryzlewicz provides a good overview of different approaches to this problem [9]. In general, there is a tradeoff between more conservative methods that fail to detect obvious changes in the series, and overly sensitive methods that detect many false positives. Our techniques work best if the identified change points are a superset of the real phase boundaries, but there is a substantial computational cost to processing spurious change points.

Our method is not tied to a specific change point detection algorithm, but we have evaluated it using a variant of binary segmentation. Binary segmentation is a classic change-point detection algorithm that initially finds, at most, a single change point and then recursively partitions the series at the change point, stopping when it can no longer find a change point. Change points are detected by measuring variations in some quantity (such as the mean) and using a test such as CUSUM [10] to determine whether the change exceeds some threshold. This algorithm is relatively intuitive and has low computational complexity ($\mathrm{O}(NlogN)$). However, it is not guaranteed to find an optimal set of change points, and it can perform poorly when the spacing between consecutive change points is short. For these reasons, we use the Wild Binary Segmentation (WBS) variant, which looks for change points in random subintervals of each partition, works better in time series with change points that are numerous or closely spaced, and allows for use of a penalty function to optimize some information criterion [9]. In this work, we use the Bayesian Information Criterion penalty function, but other standard penalty functions gave nearly identical results. We evaluate the accuracy of change point detection in Section IV-C.

B. Candidate Phase Identification

The next step is to generate *candidate phases* from pairs of change points. We then build a graph with a node for each change point, as well as for the start and end of the trace, and an edge for each candidate phase.

The minimal approach is to use only pairs of adjacent change points, which has the advantage of cleanly partitioning the trace. However, with this approach, a single spurious change point has the potential to render all subsequent phases unrecognizable.

The maximal approach is to allow every pair of change points, including the beginning and end of the trace, to be a candidate phase. This is the most computationally complex approach, since it involves applying phase recognition to each pair of points, and it also maximizes the complexity of choosing the final partition of the trace.

The results presented in this paper use the maximal approach of including all possible edges, which is feasible given the length of our traces and the number of detected change points. In other contexts, it may be necessary to prune some edges before proceeding.

C. Phase Recognition

Once the candidate phases have been identified, the next task is to match them to kernels in the training set. To do so, we train a random forest classifier [11] that can recognize individual kernels from their power traces. We follow the method of Combs et al. [1] in representing the kernels as vectors of statistical features such as mean, standard deviation, serial auto-correlation, and self-similarity. In addition to the 14 features used in that work, we also include the four highest-order coefficients, excluding the first coefficient, of a discrete Fourier transform. When broken up by real and imaginary components, we get 8 additional features. In order to directly compare DFT coefficients of different-length phases, a phase is cropped if its length exceeds some constant N. If the phase length is shorter than N, the phase is padded with zeroes. If N is large, there is a possibility that this zero-padding ties the DFT coefficients to the length of the trace, thus indirectly using the phase length as a feature for phase recognition. This may not be desirable if the goal is to recognize the workload across different hardware and input data. However, our testing indicates that length alone does not account for the increase in accuracy attributable to the DFT features, as seen in Section IV-D, Table V.

We also evaluated R*-trees [12] storing vectors of DFT coefficients, which have been used in prior work on time-series similarity [13]. However, we found that their accuracy declines precipitously if the placement of the detected change points is imperfect. Furthermore, their output does not lend itself as easily to selecting the final partition of the trace. They return a list of intersecting or nearby traces to the query trace, which then needs to be translated into a set of edge weights (see the next section).

D. Choosing Final Phase Set

Once each candidate phase has been assigned to a kernel, the final step is to select the set of candidate phases that best partitions the trace. We do this using our graph representation, where nodes are change points and edges are candidate phases. To assign each edge a weight, we use two artifacts of the random forest prediction process. The first is the *certainty* of the prediction, which is simply the fraction of the decision trees in the random forest that "voted" for the predicted workload. For our dataset, which includes traces from 21 workloads, this value will be between approximately 0.05 (if the vote is evenly split among all workloads) and 1 (if the decision trees unanimously "vote" for the same workload).

The other feature is the *proximity* of the candidate phase to the other traces in its predicted workload. Random forest classification yields an intuitive measure of the proximity of two traces, based on tracing the traces' paths through the individual decision trees that make up the random forest: if two traces are very similar, they should follow the same path through a larger percentage of the trees than two traces that are very different. The pairwise proximity is thus defined as the fraction of trees for which the two traces being compared fell into the same terminal node. We use the maximum pairwise proximity between the candidate phase and each of the other phases in its predicted workload.

To compute the final edge weight for each candidate phase, we average the certainty and the proximity (both of which are values between 0 and 1). We then multiply this product by the length of the interval represented by that candidate phase. Otherwise, a succession of many short phases with relatively low certainties and proximities could appear more attractive than a single long phase with very high certainty and proximity. The problem of choosing the final partition, then, corresponds to a longest-path search through the graph, from the node representing the start of the trace to the node representing the end of the trace.

IV. RESULTS

A. Dataset

Our training set consists of 388 power traces from 21 different kernels, as Table I shows.

TABLE I. KERNELS IN OUR 388-TRACE DATASET

Family	Name	Description
NPB [14]	bt	block tri-diagonal solver
	cg	conjugate gradient
	ft	discrete 3D FFT
	lu	Gauss-Seidel solver
	sp	scalar penta-diagonal solver
	ua	unstructured adaptive mesh
Mahout	als	alternating least-squares recommendation
	bayes	naïve Bayesian classification (2 implementations)
	sgd	stochastic gradient descent
	kmeans	k-means clustering (2 implementations)
SystemBurn [15]	tilt	
	fft1d	
	fft2d	
	dgemm	
	gups	
	scublas-dgemm	DGEMM-CPU concurrent with cuBLAS-GPU
Other	nsort [16]	external sort
	p95 [17]	Mersenne prime finder
	stream [18]	STREAM memory bandwidth benchmark
	graph500 [19]	
	baseline	active idle

For the Mahout data analytics "kernels," which are generally the most complex, we ran each kernel on at least two different input datasets of different sizes. All of those datasets are provided with the Apache Mahout download, except for LastFM360[1], which was used in ALS.

For each NPB kernel, we collected data for the serial version and the OpenMP version with 1, 4, and 8 threads, In addition, we collected data for the MPI versions of all NPB kernels except ua. For cg, ft, and lu, we collected data for 1, 4, and 8 MPI processes. The bt and sp kernels, which require a square number of processes, were run with 1 and 4 MPI processes.

The benchmarks from SystemBurn are primarily designed to stress the CPU. SystemBurn is a tool that allows the user methodically create a maximal system load for testing and validation purposes, and the benchmarks listed are among the sample workloads it provides. For both SystemBurn and the benchmarks in the "Other" category, we ran each benchmark multiple times, with at least two different data inputs, and with different configurations (e.g. number of cores) whenever possible.

We collected power traces on two single-node systems, as shown in Table II. We used system RF for the Mahout and NPB traces, and both systems for the SystemBurn and Other traces. Both machines run Linux with the *ondemand* CPU frequency scaling governor and with Turbo Boost disabled. Wall power was sampled at 1 Hz using a WattsUp? PRO power meter, whose accuracy is $\pm 1.5\%$ plus 3 counts of the reported value.

These systems hardly represent the state of the art in personal computing, let alone HPC. However, they provide a starting point for this analysis, and we hope to compare these results with data from more modern or more parallel systems. If the results are similar, it would indicate the generality of these techniques for phase detection in HPC systems. If not, it would yield insight into the factors that influence an

[1]Available at at
http://mtg.upf.edu/static/datasets/last.fm/lastfm-dataset-360K.tar.gz.

TABLE II. SYSTEMS USED IN DATA COLLECTION

Machine	LC	RF
CPU	Intel Core i5-750	Intel Core i7-3770
Cores	4	4
GHz	1.2–2.67	1.6–3.4
RAM	8 GB	8 GB
GPU	GeForce GTX 650 Ti 1GB	GeForce GTX 670 2GB
Power	85–252 W	74–309 W

application's power consumption pattern and the limits of phase detection.

B. Evaluation Procedure

For each of our repeated evaluation runs, we choose five traces at random and remove them from the training set. We concatenate the removed traces, in random order, to form a longer trace (the *test trace*). The goal is to automatically learn a model from the training set that allows us to partition the test trace into its constituent kernels. Because the concatenated kernels should come from the same machine to simulate a full application, we use only kernels from RF in the test traces. We still leave the LC traces in the training set.

One subtle but important point is how the traces are concatenated. Our traces were all collected with idle time at the beginning and end of the trace. Some of this idle time is an artifact of the measurement process in order to ensure that the power meter captured the full execution of the workload, and some of it is the ramp-up or ramp-down of the actual kernel. Unfortunately, there is not an easy way to disambiguate the two. Therefore, when we concatenate two traces, the idle time at the end of one trace blends seamlessly with the idle time at the beginning of the next. No phase detection algorithm should reasonably be expected to perfectly break what appears to be an uninterrupted idle period into two phases. We therefore do not penalize our phase detector for attributing these fused idle periods to the "baseline" kernel.

Having these semi-artificial idle phases in the test trace makes our task easier in some ways and harder in others. It makes spotting legitimate change points relatively easy, although a real application that concatenated a series of kernels would likely also have some sort of detectable ramp-down followed by a ramp-up. It is important to note that selecting the right subset of the identified change points is still nontrivial and that traces can have internal idle periods that should not be attributed to the "baseline" kernel.

C. Change Point Detection Accuracy

The first step in our method of phase detection is identifying change points in the concatenated test trace. Ideally, the change point detector will detect a superset of the actual change points that separate the concatenated kernels. If it fails to detect a true change point, we will misidentify at least some points in the trace. On the other hand, detecting a large number of false-positive change points will significantly increase the computational complexity of the remaining steps of phase detection, and it could also lead to a greater likelihood of mis-partitioning the trace.

We quantify these two situations by evaluating the precision and recall of our change point detector. The precision is the fraction of our identified change points that are true detections. The recall is the number of true detections divided by the total number of change points in the trace. A detection counts as true if it is within some distance ϵ of the actual change point.

In addition to a value of ϵ, the algorithm we use for change point detection requires a ceiling *Kmax* on the number of change points to detect. The best choice of *Kmax* depends on the complexity and length of the trace. By setting *Kmax* to the square root of the trace length, scaled by some constant S, we observed good results for our data set. Table III shows the average precision and recall over 100 runs, with S held constant at 0.5, for two different values of ϵ: a stringent value of 3, and a more lenient value of 10.

TABLE III. PRECISION AND RECALL OF CHANGE POINT DETECTION FOR DIFFERENT VALUES OF ϵ, WITH THE MAXIMUM NUMBER OF CHANGE POINTS SET TO HALF THE SQUARE ROOT OF THE TRACE LENGTH

	$\epsilon = 3$	$\epsilon = 10$
Precision	0.39	0.36
Recall	0.99	0.98

Figure 1 shows the distribution of recall and precision for $\epsilon = 3$. The recall is consistently high, with the precision more variable depending on the complexities of the concatenated traces. We also experimented with varying S. Increasing the value of S will increase the possible number of change points detected, which increases precision and decreases recall.

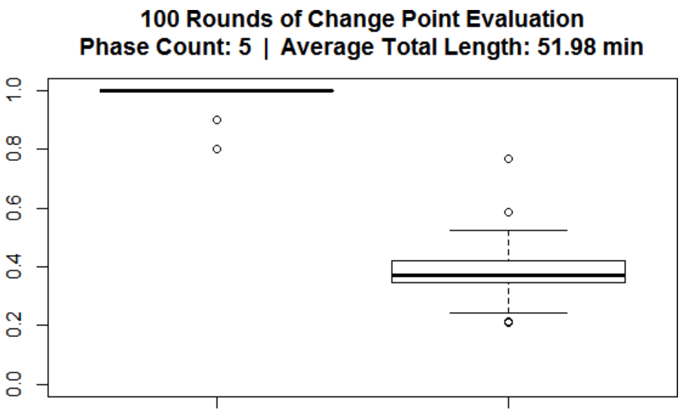

Fig. 1. Distributions of change-point detection recall (left) and precision (right) over 100 runs with $\epsilon = 3$

D. Overall Phase Detection Accuracy

Our objective is to maximize the number of data points in the test trace that are attributed to the correct kernel, a metric that covers both the placement of change points and matching phases to the correct workload. Table IV shows an example of how we calculate accuracy. This is a simplified example and not a real trace from our dataset. In this example, the test power trace consists of three concatenated kernels: A, B, and C. The phase detector correctly identifies the first phase as kernel A, but extends it by an extra 50 data points, so all 400 data points in the actual phase A were correctly detected. The phase detector also correctly identifies the second phase as B, but the endpoints of the phase are misidentified, yielding 445 correct points in phase B. Finally, the third phase is misidentified as kernel Z. Therefore, we would score this sample as 845/1000 data points correct.

TABLE IV. EXAMPLE: CALCULATING PHASE DETECTION ACCURACY ON A MADE-UP POWER TRACE WITH 1000 SAMPLES

Actual Phases		Predicted Phases		Points correct
Interval	Kernel	Interval	Kernel	
$[1, 400]$	A	$[1, 450]$	A	400
$[401, 900]$	B	$[451, 895]$	B	445
$[901, 1000]$	C	$[896, 1000]$	Z	0

Figure 2 gives a histogram of this measure of accuracy over 300 runs. The phase detector performs relatively consistently over all of the test traces, without the bimodal distribution one would expect if the phase detector often makes an early wrong decision and fails to recover. The mean accuracy is 78% over these runs.

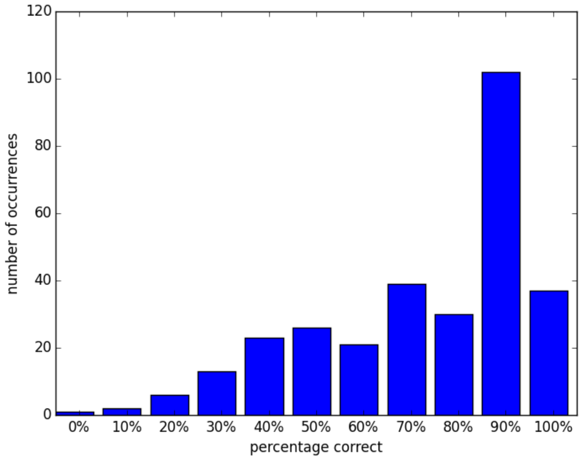

Fig. 2. Histogram of phase identification accuracies over 300 test traces

Figure 3 further illustrates that early mispredictions do not lead to unrecoverable errors later in the trace. To calculate each bar, we evaluate the accuracy over an interval of the points in the concatenated trace (the first 10% of the points, the next 10%, etc.). The distribution is relatively flat, indicating that our phase detection does not lose accuracy late in the trace. The slight increases for the first and last deciles can be attributed to the relative ease of identifying the initial and final idle phases in the concatenated trace.

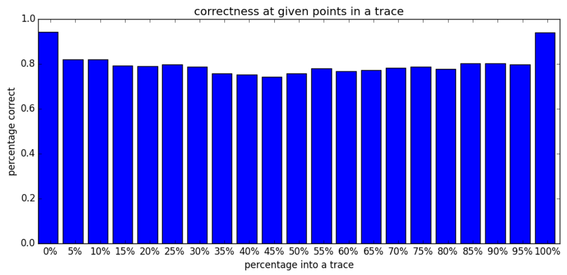

Fig. 3. Histogram of phase detection accuracy at different points in a trace

As discussed in Section III-C, in order to calculate the DFT coefficients that are used as phase recognition features, we pad or truncate a candidate phase to some value N.

TABLE V. EFFECT OF DFT COEFFICIENTS ON PHASE RECOGNITION ACCURACY WITH DIFFERENT FIXED PHASE LENGTH N

N	Accuracy	Note
not set	52%	
not set	69%	actual phase length used as additional feature
1024	78%	
2048	83%	

The question is whether the selection of N is essentially a proxy for trace length. Table V provides some reassurance on that front. Without selecting a fixed N, the DFT coefficients become a fragile basis for phase identification, with an overall accuracy of only 52%. If the actual phase length is added as a feature, the accuracy increases to 69%. However, fixing N to either 1024 or 2048 yields higher accuracy than length alone, indicating that the DFT coefficients do provide additional information helpful for recognizing the phase.

V. CONCLUSION AND FUTURE WORK

This study shows that a single power trace can be broken down into a series of kernels with high accuracy. A change point detection algorithm reliably identifies a superset of the actual transitions between kernels, which is then refined using feedback from a classifier that attempts to match intervals in the trace with known kernels.

In future work, we would like to further test the generality of this approach by varying the transition between kernels. In this work, there is an idle period between each pair of kernels, but what if the transition were more seamless, without a full ramp-down of one kernel and ramp-up of the next? Moreover, it remains to be seen how our method deals with traces consisting of a mix of known kernels with unknown kernels or noise.

It would also be useful to develop heuristics for reducing the computational complexity of our approach by pruning candidate phases when the number of identified change points is high. Finally, we would like to extend this work to more realistic HPC settings to test a wider range of configurations and datasets for each workload.

ACKNOWLEDGMENTS

This work was supported by the United States Department of Defense (DoD) and used resources of the Computational Research and Development Programs at Oak Ridge National Laboratory.

REFERENCES

[1] J. Combs *et al.*, “Power signatures of high-performance computing workloads,” in *Proc. Energy Efficient Supercomputing Workshop (E2SC)*, 2014.

[2] A. S. Dhodapkar and J. E. Smith, “Comparing program phase detection techniques,” in *Proc. of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2003.

[3] G. L. T. Chetsa, L. Lefèvre, J.-M. Pierson, P. Stolf, and G. Da Costa, “A runtime framework for energy efficient HPCw systems without a priori knowledge of applications,” in *Proc. 18th International Conference on Parallel and Distributed Systems (ICPAD)*, 2012.

[4] I. Paul, V. Ravi, S. Manne, M. Arora, and S. Yalamanchili, “Coordinated energy management in heterogeneous processors,” in *Proc. International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

[5] N. Ioannou, M. Kauschke, M. Gries, and M. Cintra, “Phase-based application-driven hierarchical power management on the single-chip cloud computer,” in *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011.

[6] J. Peraza, A. Tiwari, M. Laurenzano, L. Carrington, and A. Snavely, “PMaC’s Green Queue: A framework for selecting energy optimal DVFS configurations in large scale MPI applications,” *Concurrency and Computation: Practice and Experience*, 2012.

[7] C. Isci and M. Martonosi, “Identifying program power phase behavior using power vectors,” in *Proc. IEEE Int. Workshop on Workload Characterization (WWC)*, 2003.

[8] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos, “Fast subsequence matching in time-series databases,” in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1994, pp. 419–429.

[9] P. Fryzlewicz, “Wild binary segmentation for multiple change-point detection,” *Annals of Statistics*, vol. 42, no. 6, pp. 2243–2281, 12 2014.

[10] E. S. Page, “Continuous inspection schemes,” *Biometrika*, vol. 41, no. 1-2, pp. 100–115, 1954.

[11] L. Breiman, “Random forests,” *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[12] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, “The R*-tree: An efficient and robust access method for points and rectangles,” in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1990, pp. 322–331.

[13] T. Kahveci and A. K. Singh, “Optimizing similarity search for arbitrary length time series queries,” *IEEE Trans. on Knowl. and Data Eng.*, vol. 16, no. 4, pp. 418–433, Apr. 2004.

[14] 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, “The NAS Parallel Benchmarks&mdash;summary and preliminary results,” in *Proc. ACM/IEEE Conference on Supercomputing (SC)*, 1991.

[15] J. Lothian *et al.*, “SystemBurn: Principles of design and operation release 3.1,” Oak Ridge National Laboratory, Tech. Rep. ORNL/TM-2013/234, 2013.

[16] C. Nyberg *et al.*, “Nsort,” Ordinal Technology Corp. [Online]. Available: http://www.ordinal.com/

[17] “Great Internet Mersenne prime search.” [Online]. Available: http://www.mersenne.org

[18] J. D. McCalpin, “STREAM: Sustainable memory bandwidth in high performance computers.” [Online]. Available: http://www.cs.virginia.edu/stream/

[19] “The Graph500 list.” [Online]. Available: http://www.graph500.org/