# A Case for Criticality Models in Exascale Systems

Brian Kocoloski*, Leonardo Piga†, Wei Huang†, Indrani Paul†, and John Lange*
* Department of Computer Science
University of Pittsburgh
{briankoco, jacklange}@cs.pitt.edu
† AMD Research
Advanced Micro Devices, Inc.
{Leonardo.Piga, WeiN.Huang, Indrani.Paul}@amd.com

***Abstract*—*Performance variation* is a significant problem for large scale HPC systems and will increase on future exascale systems. In this work, we show that performance variation impacts the performance and energy efficiency of contemporary large-scale computing systems in highly temporally inconsistent ways. We thus present a case for criticality models, a learning based mechanism that allows a system to generate holistic models of performance variation as it occurs during application runtime. Criticality models are designed to provide a mechanism by which applications can detect performance variation at runtime and take action to mitigate its effects. We present a promising preliminary analysis of criticality models on a small scale cluster. Our results demonstrate that models based on logistic regressions can accurately model criticality at this scale.**

***Index Terms*—performance variation; exascale computing; criticality modeling; performance modeling**

## I. Introduction

*Performance variation* is a significant problem for large scale HPC systems. Given the tightly synchronized nature of HPC applications, execution time and energy efficiency are dictated by the slowest ranks ("stragglers") in the system. To date, it is commonly assumed that the stragglers are made up of a consistent subset of nodes [1], [2]. However, in emerging HPC systems, variation can be highly temporally inconsistent [3], meaning not only will some components of the application be more impacted than others, but also the nodes in the straggler group will change over time. Such inconsistency is driven by issues that are external to the application and usually out of its control, such as resource contention [4] and operating system (OS) interference [5]. Furthermore, it is widely expected that exascale systems will feature multiple consolidated workloads on single nodes [6] and allocate power non-uniformly between nodes [7], [8]. These characteristics combined with unprecedented scale will likely lead to temporally inconsistent behavior.

On future exascale systems, a single top-down approach to *eliminate* variation will be challenging if not impossible to achieve. Therefore, instead of preventing it, researchers have begun to explore ways to react and adapt to it as it occurs; for example, non-uniform power distribution [1], [2], [3], and workload redistribution [9], [10] have been shown to improve runtime and/or energy efficiency in the face of variation. However, these approaches generally assume relatively simple models of variation that will not capture the complexity of temporally inconsistent variation.

This paper presents a case for *criticality models*, a learning based mechanism that allows a system to generate holistic

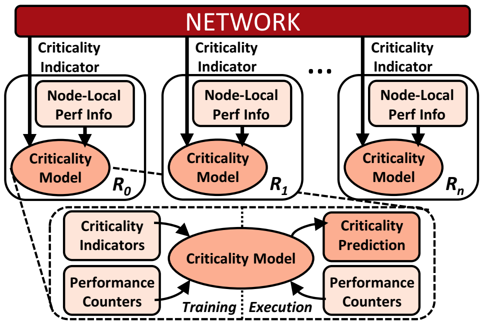

Fig. 1: A high-level view of criticality models

models of performance variation as it occurs during application runtime. Criticality models use low-level performance measurements and system-wide criticality labels based on application timing variation in order to learn how variation manifests itself, allowing it to be quickly and easily measured by a node. Once generated, criticality models can be used at any time to estimate how critical a node's performance is to the progress of the application as a whole.

A high-level view of our criticality modeling approach is shown in Figure 1. Each rank contains an internal criticality model used to predict its likelihood of being critical to the application's progress (i.e. being one of the stragglers). By grouping ranks based on their criticality predictions, the models collectively produce a *critical set* of ranks whose members are most likely to delay progress. Most importantly, once generated, criticality models require no communication between ranks – each rank executes its model independently and makes its own predictions of critical set membership. As a result, criticality models are more scalable than approaches that require frequent communication to determine criticality. Furthermore, because criticality models are agnostic to sources of variation, they are well-suited to address the myriad sources that will impact extreme scale systems, including complex causes of temporally inconsistent variation.

## II. Temporally Inconsistent Variation

Performance variation significantly impacts the runtime and energy efficiency of large scale systems. This paper claims that performance variation cannot be modeled using simple and static classifications of only *spatially* inconsistent variation as commonly assumed. This is motivated by the primary finding we make in this section that refutes a commonly held

| Application | # Ranks | # Ranks always or sometimes "slow" | | | |
|---|---|---|---|---|---|
| | | "Slow" == Last 5% | | "Slow" == Last 50% | |
| | | Always | Sometimes | Always | Sometimes |
| AMR Boxlib | 13,824 | 0 | 13,824 | 0 | 13,824 |
| BigFFT | 1,024 | 1 | 205 | 118 | 940 |
| FillBoundary | 10,648 | 0 | 1,743 | 840 | 9,770 |
| MiniFE | 1,152 | 0 | 644 | 0 | 1,152 |
| MultiGrid | 10,648 | 0 | 4,784 | 6 | 10,643 |

TABLE I: Temporal inconsistency of performance variation

assumption: performance issues are not always constrained to the same set of "straggler" ranks over the lifetime of an application, but rather move between different nodes of the system over time.

To understand how performance variation affects applications at extreme scale, we first analyze the performance of a set of (mini-)applications provided by the US Department of Energy (DOE)[1]. The traces are for a set of applications that characterize DOE requirements and can provide insight into the effects of application imbalance and the behavior of stragglers in today's large scale systems. We analyze AMR Boxlib and MultiGrid, full traces of production applications used by the DOE, and BigFFT, FillBoundary and MiniFE, kernels extracted from corresponding production applications.

Table I demonstrates temporal variation inconsistency by investigating the lack of consistently slow ranks in the applications. It considers two different definitions for a "slow" rank: a rank that is one of the last 5% of ranks to reach a global collective or a rank that is on the last 50% to reach a collective. When considering the slowest 5% of ranks, the results show that classifying a single set of ranks as consistently "slow" is not possible. Moreover, a substantial number of ranks in AMR Boxlib (100%), MiniFE (56%) and MultiGrid (45%) will eventually belong to the "slow" set (as can be seen in the *Sometimes* column). Similar results hold for the 50% threshold, where in all cases over 90% of ranks will fall into the "slow" category at some point in time. These results show significant temporal performance variation for a given rank across the lifetime of the applications. Static classifications of a rank's criticality that do not change over time are inappropriate for these systems.

## III. A Case for Criticality Models

Performance variation is a major impediment to the performance of current large scale HPC systems. Unfortunately, there are indications that performance variation will be even more of an issue at exascale [6], [7], [8], [11]. Based on the sheer number and complexity of sources that will induce variation, a top-down approach to eliminating all sources will be challenging to achieve. Thus, we propose *criticality models*, a learning based mechanism that allows the system to generate holistic models of performance variation as it occurs during application runtime. Criticality models use observations of how performance variation manifests across a large scale system to automatically learn the local node-level performance characteristics that indicate criticality. Criticality models are designed to provide a mechanism by which applications and runtime systems can detect and react to variation in an intelligent manner rather than preventing performance variation from occurring.

### A. Machine Learning Based Model Training

Due to the complex nature of variation at exascale, generating heuristic based models that can capture all sources of variation is challenging. Instead, we propose the use machine learning to generate models. The key advantage of this approach is that features indicative of variation can be automatically learned and selected during the model generation phase.

Machine learning techniques can generate accurate models for a wide range of applications and system architectures. Given that different applications are sensitive to different types of performance variation, a "one size fits all" model cannot perform well for all combinations of applications and systems. Thus, machine learned models can be superior in terms of model performance as well as human effort in generating tailored models for new applications and systems whose performance characteristics may not be easily understood.

### B. Distributed Model Generation

Criticality models are generated in a distributed fashion. Each node in the system is responsible for periodically collecting low-level performance measurements as an application is running. These measurements may take the form of hardware performance counters, software events (e.g., percentage of time context switching, breakdown of user mode vs. kernel mode time), and/or counters related to the network (e.g., bytes per second received at a network endpoint over an interval of time). Measurements can also contain application-level information, such as the most recent MPI call made by a rank.

Models can be generated from these measurements either online or offline. In either case, the key is that each rank-specific performance measurement must be labeled with an indicator of criticality to feed to the machine learning phase. For online generation, each rank labels its performance measurements when it reaches the next global communication point. Labels are based on the rank's communication time (e.g., MPI slack) in comparison to the communication time of all other ranks; i.e., ranks that are comparatively slower than most other ranks have measurements annotated "critical;" i.e., they belong to the *critical set*. Such a critical set is likewise defined for every global collective in the application.

### C. Parallel and Autonomous Model Execution

Once criticality models have been generated, each node executes its model by collecting node-local performance measurements and generating a local prediction of criticality based on these measurements. The main benefit of this approach is a very high degree of scalability – nodes do not need to communicate with each other to derive predictions of their own criticality. This design is based on the observation that, as systems scale to potentially millions of nodes, requiring

[1]Detailed system and application specifications can be found: http://portal.nersc.gov/project/CAL/designforward.htm

inter-node communication to execute a criticality model will be inherently unscalable, particularly if the results of the prediction need to be generated in a timely manner (e.g., redistribute power before the next collective). Our criticality models eschew global communication, instead allowing each node to predict its criticality based only on measurements that it can quickly and easily collect as the application is running.

## IV. Preliminary Evaluation

In this section, we show how criticality models can be built and executed on trace runs from MPI applications on a small scale cluster. While criticality models target variation on large scale systems, our initial experimental analysis focuses on demonstrating the effectiveness of the models in identifying the low level indicators of criticality.

### *A. Performance Measurements and Profiling*

Criticality models are generated from low-level performance measurements collected independently by each rank. For this analysis, we have limited performance measurements to hardware performance counters. Models are first trained with measurements from ranks deemed to be "critical" (i.e., in the critical set) or "not critical" (i.e., not in the critical set). These ground truth annotations are provided in an offline manner by post-processing MPI traces generated from application runs, using MPI slack via the DUMPI [12] library.

We collect performance counters using the Instruction Based Sampling (IBS) mechanism found in modern AMD processors [13]. IBS-enabled processors are configured to generate interrupts after a configurable number of cycles (or instructions) has elapsed and provide information about how the instruction was processed in the core's pipeline (e.g., TLB and cache hit rates, number of cycles elapsed since the instruction was issued, etc.). We collected over 20 different performance measurements from each IBS instruction.[2]

### *B. Constructing the Models*

We construct criticality models in an application specific manner. We select a set of mini-applications from the Mantevo suite [14] (CloverLead, HPCCG, and MiniFE), as well as two real applications (AMR Boxlib [15] and ParaDiS [16]. All models are trained offline from application traces collected on a small 4-node experimental cluster. Each node has a 4-core AMD A10-7850K processor with 16 GB RAM, with each node interconnected with MT27600 Mellanox InfiniBand cards. Each application is compiled against the DUMPI library to generate detailed MPI traces. Each rank is also configured to collect IBS performance counters during its execution.

The key parameter guiding the construction of a criticality model is the *imbalance* observed at each collective in an application. We define the per-collective *imbalance* as the maximum MPI slack across all ranks for a given collective. Formally, if $R$ is the set of all ranks, $K$ the set of all collectives, $S_{r,k}$ the MPI slack experienced by rank $r$ at collective $k$, $I_k$ the imbalance at collective $k$, then the annotation for rank $r$'s performance counters at collective $k$, $PC_{r,k}$, is as follows:

$$\forall r \in R, k \in K \text{ s.t. } I_k >= 10 \text{ ms}, PC_{r,k} = \begin{cases} \text{“Critical”} & \text{, if } S_{r,k} < (I_k * 0.25) \\ \text{“Not critical”} & \text{, otherwise} \end{cases} \quad (1)$$

As stated in Equation 1, ranks whose slack is less than a quarter of the collective's imbalance (slack of the fastest rank) have their performance measurements classified as belonging to the critical set, while all others have their counters classified as belonging to a not critical rank. Currently, the criticality models are generated with a logistic regression, which takes a single IBS measurement as input and predicts whether the instruction was generated from a critical or non-critical rank. We plan to investigate other, more complex classification tools (support vector machines, neural networks, etc.) in the future. We choose 10 ms as a threshold for determining which collectives exhibit a large degree of imbalance and thus should be used as input to the model generation phase.

### *C. Evaluating the Models*

Having constructed the models, we evaluate them using additional trace data. After collecting performance counters for 5 ms after a collective, each rank generates a criticality prediction that remains in effect until the next collective is reached. Practically speaking, this means that predictions are not made for computational periods (between MPI collectives) shorter than 5 ms; thus, we do not evaluate the models on these periods. This value still allows criticality predictions to be generated quickly enough to be effective in reacting to performance variation, but is long enough to generate a representative view of the performance of a node. We omit a detailed evaluation of this parameter due to space constraints.

To evaluate the models, we perform a series of cross-validations. Our goal in these evaluations is not only to determine how well variation can be modeled by performance counters, but also whether the absolute values of imbalance have any effect on accuracy. Each application is executed three times to generate data sets for training and testing. The input data for each application (performance counters and criticality annotations) is aggregated from these runs and then randomly sampled without replacement such that 75% of the data is used for training and the remaining 25% for testing. We then perform $k$-fold cross-validation with $k$=10.

Figure 2 shows the results of the cross-validations. An individual classification is deemed accurate if it matches the annotation for the rank which generated the performance counter ("critical" or "not critical"). The results are partitioned based on the imbalance seen in the collective that the sample was generated in. For example, the leftmost bar shows the classification accuracy for all samples in AMR Boxlib that were generated from a collective with at least 5 ms but less than 10 ms of imbalance. Missing bars indicate that no periods with the specified imbalance occurred in the application.

[2]The full list of IBS performance measurements can be found on page 597 of the AMD BKDG [13]

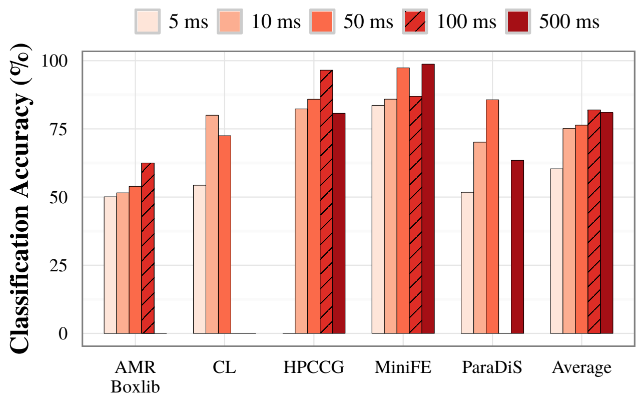

Fig. 2: Classification accuracies of criticality models

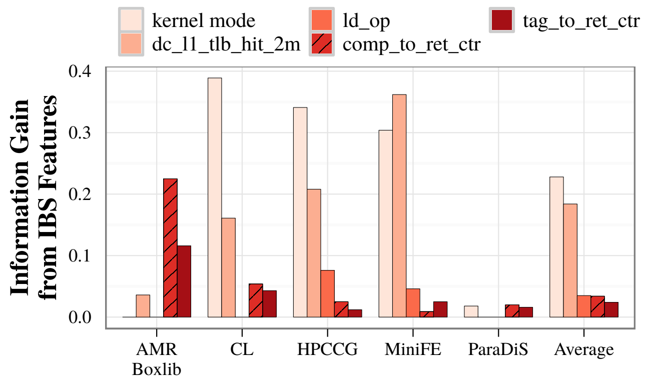

Fig. 3: Information gain from IBS performance events

In general, larger levels of imbalance result in better classification accuracy. This suggests that the models are able to better identify predictive characteristics from performance counters that are generated during periods of relatively large imbalance and that imbalance does indeed manifest in the node-level performance discrepancies for these applications. AMR Boxlib is the only application whose criticality model classified with less than 60% accuracy, unless the sample came from a collective with at least 100 ms of imbalance. Additional performance metrics such as network performance counters or MPI call stacks may be able to better explain imbalance in this application and we plan to investigate such metrics.

Finally, we also determine which types of performance counters gleaned from IBS sampling are most influential in generating the criticality models. Figure 3 shows the information gain from the five most influential performance events in the IBS measurements. The figure shows that the two most indicative signs, on average, are whether a sample is generated from kernel space (`kernel mode`) and whether the physical address for a tagged load/store is in a 2MB page table entry in data cache L1 TLB (`dc_l1_tlb_hit_2m`). An instruction being a load operation (`ld_op`) provides insight for some applications (HPCCG and MiniFE). Finally, the number of cycles from instruction completion to retirement (`comp_to_ret_ctr`) and from instruction tagging to retirement (`tag_to_ret_ctr`) also have predictive influence.

## V. Conclusion

Performance variation will be a significant impediment to the runtime and energy efficiency of future HPC systems. We introduced criticality models to address the increasing complexity caused by temporally inconsistent variation. Criticality models are designed to learn how causes of variation manifest across a system and provide a scalable, low latency mechanism to inform higher level services how and where variation occurs. We presented a promising early evaluation of criticality models on a small cluster, showing that logistic regressions can accurately model criticality at this scale.

## References

[1] B. Rountree *et al.*, “Adagio: Making DVS Practical for Complex HPC Applications,” in *Proc. 23rd ACM International Conference on Supercomputing*, ser. (ICS), 2009.
[2] P. Bailey *et al.*, “Finding the Limits of Power-Constrained Application Performance,” in *Proc. 27th Annual IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. (SC), 2015.
[3] A. Venkatesh *et al.*, “A Case for Application-oblivious Energy-efficient MPI Runtime,” in *Proc. 27th Annual IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. (SC), 2015.
[4] A. Bhatele *et al.*, “There Goes the Neighborhood: Performance Degradation due to Nearby Jobs,” in *Proc. 25th Annual IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. (SC), 2013.
[5] T. Hoefler *et al.*, “Characterizing the Influence of System Noise on Large-Scale Applications by Simulation,” in *Proc. 22nd Annual IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. (SC), 2010.
[6] F. Zheng *et al.*, “GoldRush: Resource Efficient In Situ Scientific Data Analytics Using Fine-Grained Interference Aware Execution,” in *Proc. 25th Annual IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. (SC), 2013.
[7] O. Sarood *et al.*, “Optimizing Power Allocation to CPU and Memory Subsystems in Overprovisioned HPC Systems,” in *Proc. 15th IEEE International Conference on Cluster Computing*, ser. (CLUSTER), 2013.
[8] T. Patki *et al.*, “Exploring Hardware Overprovisioning in Power-Constrained, High Performance Computing,” in *Proc. 27th ACM International Conference on Supercomputing*, ser. (ICS), 2013.
[9] J. Dinan *et al.*, “Dynamic Load Balancing of Unbalanced Computations Using Message Passing,” in *Proc. 6th International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems*, ser. (PMEO-PDS), 2007.
[10] B. Acun *et al.*, “Mitigating Processor Variation through Dynamic Load Balancing,” in *Proc. 1st IEEE International Workshop on Variability in Parallel and Distributed Systems*, ser. (VarSys), 2016.
[11] T. Patki *et al.*, “Practical Resource Management in Power-Constrained, High Performance Computing,” in *Proc. 24th International ACM Symposium on High-Performance Parallel and Distributed Computing*, ser. (HPDC), 2015.
[12] Sandia National Laboratories, “Using DUMPI,” online, 2015, http://sst.sandia.gov/using_dumpi.html, Accessed on 16 April 2015.
[13] Advanced Micro Devices, Inc., “BIOS and Kernel Developers Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors,” online, 2015, http://support.amd.com/TechDocs/42301_15h_Mod_00h-0Fh_BKDG.pdf.
[14] M. Heroux *et al.*, “Improving Performance via Mini-applications,” Sandia National Laboratories, Tech. Rep., 2009.
[15] P. Colella *et al.*, *Chombo Software Package for AMR Applications - Design Document*, 2013.
[16] V. Bulatov *et al.*, “Scalable Line Dynamics in ParaDiS,” in *Proc. 16th Annual IEEE/ACM International Conference for High Performance Computing, Storage, Networking and Analysis*, ser. (SC), 2004.